AD-A104 587 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2

COMPUTER PERFORMANCE PREDICTION OF A DATA FLOW ARCHITECTURE.(U)

JUN 81 D J HOGEN

UNCLASSIFIED NL

1 of 1

AD A104587

END

DATE FILMED

10 81

DTIC

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

COMPUTER PERFORMANCE PREDICTION
OF A
DATA FLOW ARCHITECTURE

by

David John Hogen

June 1981

Thesis Advisor: L. A. COX, Jr.

Approved for public release; distribution unlimited

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. A104 587
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) Computer Performance Prediction of a Data Flow Architecture	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis, June 1981
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) David John Hogen	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	12. REPORT DATE June, 1981
	13. NUMBER OF PAGES 91
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Data Flow   Computer Performance Evaluation
Dataflow   Computer Performance Prediction

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Supercomputers capable of performing extremely high speed computation have been proposed which are based on an architecture known as data flow. Application of a Petri net-based methodology is used to evaluate the performance attainable by such an architecture. The architecture evaluated is MIT's cell block data flow architecture which is being developed to execute the applicative programming language VAL.

DD FORM 1 JAN 73 1473   EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Results show that for the data flow architecture to achieve its goal of high speed computation, intelligent multiprogramming schemes need to be developed. One such scheme, based on the notion of a "concurrency vector", is introduced.

Accession For	
NTIS GRA&I	✓
DTIC TAB	
Unannounced	
Justification	
By	
Distribution/	
Availability Codes	
Dist	
A	

DD Form 1473
1 Jan 73
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Approved for public release; distribution unlimited.

Computer Performance Prediction of a Data Flow Architecture

by

David John Hogen
Lieutenant Commander, United States Navy
B.S.O.E., United States Naval Academy, 1972

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1981

Author: David J. Hogen

Approved by: Lyle A. Cox Jr.
Thesis Advisor

Bruce J. MacLennan
Second Reader

Chairman, Department of Computer Science

W M Woods
Dean of Information and Policy Sciences

ABSTRACT

Supercomputers capable of performing extremely high speed computation have been proposed which are based on an architecture known as data flow. Application of a Petri net-based methodology is used to evaluate the performance attainable by such an architecture. The architecture evaluated is MIT's cell block data flow architecture which is being developed to execute the applicative programming language VAL.

Results show that for the data flow architecture to achieve its goal of high speed computation, intelligent multiprogramming schemes need to be developed. One such scheme, based on the notion of a "concurrency vector", is introduced.

TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

Despite the orders-of-magnitude increase in computation speed that has occurred since the early 1950's, the need still exists today for faster computers. This need is most critical in the area of scientific computing, where there exist computations requiring on the order of a billion floating point operations per second [DENNIS, 1980].

One approach to achieving higher computation speed is to increase the speed of the basic logic devices of the computer. This approach, effective in the past, faces significant obstacles to future gains because of the speed of light limitation to signal propagation and limitations in the integrated circuit manufacturing process.

A second approach to achieving higher computation speed is through the exploitation of parallelism which is (or can be) inherent in algorithms used to solve a wide range of scientific problems. Such parallelism can be present at both the operation and procedure levels in a program. Thus far, such exploitation of parallelism has not reached a limiting threshold to faster computation.

Data flow computing has been proposed as a conceptually viable method of achieving higher computational speed through greater exploitation of inherent algorithmic

parallelism. A computer based on the data flow concept executes an instruction when its operands become available. No sequential control flow notion exists. Data flow programs are free of sequencing constraints except those imposed by the flow of operands between instructions. Thus, a data flow computer contrasts fundamentally with the "von Neumann" model. Even so, the data flow concept is capable of incorporating into one system all the known forms of parallelism exploitation including vectorization, pipelining, and multiprocessing.

B. RESEARCH APPROACH

It was the purpose of this research to gain insights into the degree of parallelism exploitation obtainable with the data flow-based high speed computation method. The classical issues of hardware utilization, program execution time, and "degree" of multiprogramming were investigated in the context of data flow. Application of an existing Petri net-based methodology was the technique used to gain these insights.

The hypothesis for this research had two parts. First, the suitability of the Petri net-based Requester-Server methodology for prediction of the performance of data flow machines in an efficient, accurate manner was to be explored. Second, a challenge to the data flow concept was made. It was hypothesized that the goal of achieving higher

speed computation through data flow computing is unattainable without achieving a high and "intelligent" degree of multiprogramming. By "intelligent" it is meant that the mapping of processes onto the hardware shall have to be done in a near optimal fashion, defined in terms of hardware utilization and program execution time.

To explore the two-part hypothesis, sets of Petri net models of data flow programs, characterized by a range of inherent parallelism, were "executed" on Petri net models of the Dennis-Misunas data flow hardware design [DENNIS, AUG 1974], using the methodology called Requester-Server [COX, 1978]. The hardware models were varied in the number of processing elements available for use in "executing" the sets of program models. Thus software models were "run" on hardware models, and appropriate performance indices were measured and analyzed.

This research is important because it suggests a method for mapping data flow programs onto the data flow machine to achieve the desired degree of high speed computation. Additionally, the Requester-Server (R-S) methodology has been shown to be an effective tool for predicting the performance of data flow computer architectures.

Grateful acknowledgment is made to L. A. Cox, designer and initial implementer of the R-S methodology, and to D. M. Stowers, who modified the R-S software to enable it to run on the PDP-11/50 minicomputer at NPS.

C. ORGANIZATION

The results reported here are organized in a fashion conducive to the communication of experimental computer science endeavors. Following a review of the applicable literature (Section II), the hypothesis (Section III) is presented. Next, the method used to test the hypothesis is presented in detail (Section IV). This section discusses the experimental design which includes the identification of independent and dependent variables, characterizes and explains the Petri net definition of the data flow hardware and software, and ends with an account of the procedure used to implement the experiment to test the hypothesis. Results of the experiment and a discussion thereof are covered in Section V. This section, in addition to demonstrating the suitability of the R-S technique, and exploring the multiprogramming response of data flow architectures, presents some unexpected findings. Section VI summarizes the entire research effort, including the results. Finally, Section VII presents recommendations for further investigation in the area of data flow research.

II. LITERATURE REVIEW

A. APPROACHES TO PARALLELISM

In general, computer science literature approaches the concept of parallelism exploitation from either an architecture (hardware) or language (software) point of view. In this thesis, "parallelism" shall be viewed as existing at many hierarchical levels within algorithms. Any of several different computer architectures may be capable of exploiting the parallelism which exists at one or more of these various hierarchical levels. As should be expected, each architecture is best-suited at exploiting inherent algorithmic parallelism at a particular hierarchical level, but not at others. In contrast, implementation of the data flow concept proposes to exploit inherent algorithmic parallelism at all hierarchical levels, in an efficient fashion. Before presenting the concept of data flow, a review of the range of architectures and strategies currently used to exploit parallelism is presented.

In an early study of high speed computer architectures [FLYNN, 1966] a four element taxonomy was developed which classified computer systems in terms of the amount of parallelism in their instruction streams and data streams (see figure II.A.1). (An instruction stream is the series of operations used by the processor; a data stream is the

series of operands used by the processor.) The first element of this taxonomy, depicted in figure II.A.1(a), is the serial computer which executes one instruction at a time, affecting at most one data item at a time. Such a serial machine is denoted as a single-instruction single-data-stream (SISD) computer. The SISD computer can be characterized as possessing no capability for exploitation of algorithmic parallelism. The three remaining computer system organizations within the Flynn taxonomy do possess capabilities for exploiting algorithmic parallelism.

By allowing more than one data stream a single-instruction multiple-data-stream computer results, as shown in figure II.A.1(b). This organization allows vectorization and is known as a vector or array processor because each instruction operates on a data vector during each instruction cycle, rather than on just one operand. The model in figure II.A.1(b) shows N processors each accepting as input its own data stream. It is noteworthy that each of the N processors is not a standalone serial machine (SISD computer) because the N processors take the same instruction from an external control unit at each time step.

If the SISD computer is extended to permit more than one instruction stream, the multiple-instruction single-data-stream (MISD) computer shown in figure II.A.1(c) results. This computer system organization within Flynn's taxonomy is present for completeness but has yet to be shown

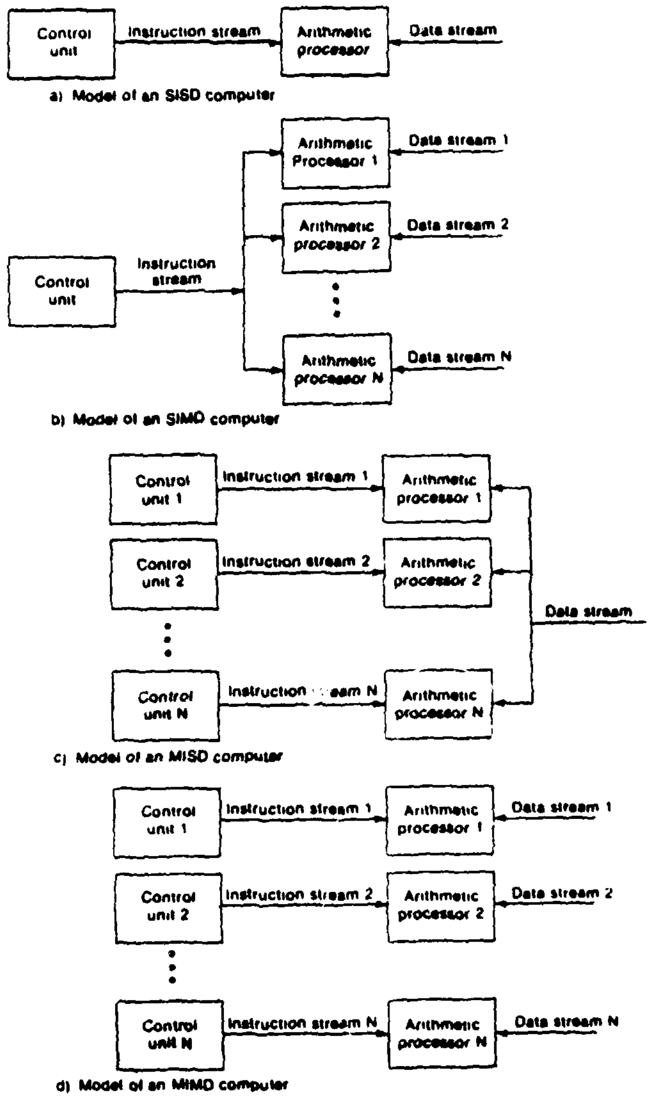


Figure II.A.1: COMPUTER MODELS [STONE, 1980]

to possess much utility. An example of such a machine would be one built to generate tables of functions (such as squares and square roots) of a stream of numbers. Each processor would perform a different function on the same data item at each time step.

The fourth and final element of Flynn's taxonomy is one which possesses parallelism in both the instruction and data streams. This multiple-instruction multiple-data-stream (MIMD) computer (shown in figure II.A.1(d)) is made up of N complete SISD machines which are interconnected for communication purposes. Such a parallel architecture is more readily recognized as a multiprocessor in which as many as N processors can be performing useful work at the same time.

Beyond Flynn's taxonomy are other approaches to parallelism. The first, pipelining, is a strategy which makes use of the fact that a processor, in executing an instruction, actually performs a sequence of functions in various functional units of the processor. Each function is performed at a different stage along the pipeline. Figure II.A.2 shows a processor with a simple pipeline design. Rather than waiting for each instruction to be completely executed before beginning the next instruction, the pipeline processor begins execution of the next instruction as soon as functional units at the beginning of the pipeline are available. Thus, the pipeline is normally full, containing more than one instructions in various stages of execution.

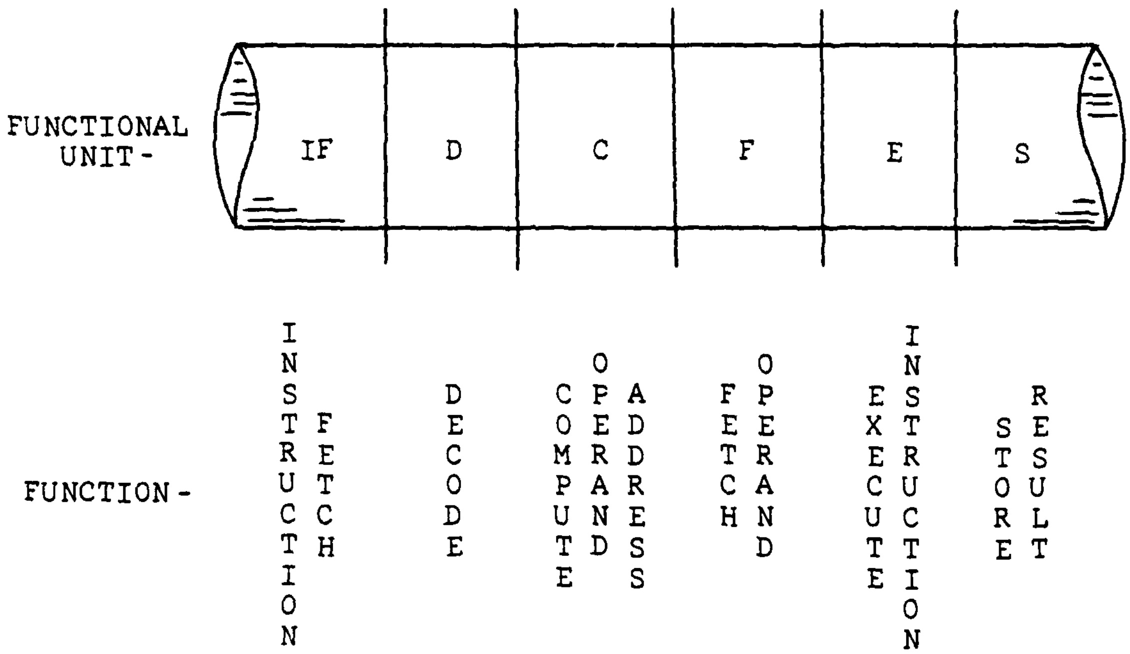


Figure II.A.2: PIPELINING

The final approach to parallelism to be presented is the strategy of overlapping. In the traditional sense, overlapping within a computer system occurs when the central processing unit (CPU) is allowed to function concurrently with input/output (I/O) operations. Such concurrency was prevented in early computers because I/O operations required data paths to memory which ran through CPU registers, preventing CPU functions from occurring while performing I/O. Overlapping can occur in other ways within a computer but the example given is sufficient to convey the general idea.

The techniques for exploiting algorithmic parallelism that have been presented are not all mutually exclusive. For example, the strategies of pipelining and overlapping can be included in any of the four architectures. Furthermore, other more complex machine organizations have been proposed. One example is the multiple SIMD (MSIMD) machine which consists of more than one control units sharing a pool of processors through a switching network [HWANG, 1979]. Such hybrids will not be considered further.

Having described the major architectural approaches to exploiting algorithmic parallelism it is appropriate to characterize the problems for which each method is suitable and to present some of the difficulties that still exist in using each method. By "suitable" it is meant that the method allows the processing of a problem in such a manner that

some speedup in execution time is achieved in comparison with what the execution time would be for the problem run on a serial machine.

The main implementation of the SIMD architecture, the array processor, is suitable for computations which can be described by vector instructions. Also, operands processed simultaneously must be capable of being fetched simultaneously from memory. Finally, processor interconnections must support high speed data routing between processors. If any of the above conditions are not met, then the computation may execute in a predominantly serial fashion within this SIMD computer. Because of these required conditions, the array processor is generally considered to be a specialized, rather than general purpose, machine.

As previously mentioned, the MISD architecture exists merely to complete the Flynn taxonomy and will not be discussed further [STONE, 1980].

The dominant MIMD computer is the multiprocessor. The multiprocessor is considered to be a general purpose machine. Accordingly, many problems should be well-suited for execution on such an architecture. Despite the fact that such systems have been shown to work well in a number of applications, especially those which consist of a number of concurrently-processable subproblems with minimal data sharing, numerous questions have yet to be answered. These

questions include how to best organize the parallel computations (i. e. partition the problem) to optimize the use of the cooperating processors, how to syncnronize the processors in the system, and how to best share the data among system processors. Also, problems which possess an iterative structure can run efficiently on an array processor and avoid the overhead of synchronization and scheduling required of the multiprocessor [STONE, 1980].

Although not considered "architectures" in the sense of Flynn's taxonomy, both pipelining and overlapping (of which there exist different types) are general purpose strategies that can be applied to most problems. Like the multiprocessor, these techniques also permit the partitioning of a problem so that several operating hardware pieces can function concurrently. Accordingly, pipelining and overlapping are often considered to be forms of multiprocessing. The difference lies in the fact that pipelining and overlapping perform partitioning at different hierarchical levels of a problem than does the multiprocessing technique.

Armed with an understanding of the diverse architectural approaches to parallelism exploitation that have been used to date, it is logical to proceed with an alternative approach, that of data flow. Before doing so, however, Petri nets will be introduced. This is appropriate because the

concepts of data flow computation are a direct application of Petri net theory.

B. PETRI NETS

Petri net theory plays an important part in this research endeavor for two reasons. First, as has been mentioned, Petri net theory forms the basis for the concepts used to describe and define data flow computation. Second, Petri net theory is the basis for the Requester-Server methodology that is used in this thesis research as a computer performance prediction tool. Because of its applicability, Petri net theory shall be presented herein, in an informal manner, with emphasis placed on its use in modelling parallel computation. Those desiring a more formal and complete discussion of Petri nets are referred to [PETERSON, 1977].

Petri nets may be thought of as formal, abstract models of information flow. Their main use has been in the modelling of systems of events in which some events may occur concurrently but there exist constraints on the frequency, precedence and concurrence of these events. A Petri net graph models the static structure of a system. The dynamic properties of a system can be represented by "executing" the Petri net in response to the flow of information (or occurrence of events) in the system.

The static graph of a Petri net is made up of two types of nodes: circles (called places) which represent conditions, and bars (called transitions) which represent events. These nodes are connected by directed arcs running from either places to transitions or transitions to places. The source of a directed arc is the input, and the terminal node is the output. The position of information in a net is represented by markers called tokens.

The dynamic execution of a Petri net is controlled by the position and movement of the tokens. A token moves as a result of a transition firing. In order for a transition to fire, it must be enabled. A transition is enabled when all of the places which are inputs to a transition are marked with a token. Upon transition firing, a token is removed from each of the input places and a token is placed on each of the output places of the transition. Thus, in modelling the dynamic behavior of a system, the occurrence of an event is represented by the firing of the corresponding transition.

Figures II.B.1 through II.B.4 show a Petri net at progressive stages of execution. As can be observed, the status of the execution at a given time can be described by the distribution of the tokens in the net. This distribution of tokens in a Petri net is called the net marking and uniquely defines the state of the net for any given instant.

Petri nets are uninterpreted models. Thus, some significance must be attached to token movement to indicate the intent of the model. This is usually done by labelling the nodes of a net to correspond in some way to the system being modelled. However, it should be remembered that the labelling of the nodes of a Petri net in no way affects its execution. A second attribute of Petri nets is their ability to model a system hierarchically. An entire net may be replaced by a single node (place or transition) for modelling at a greater level of abstraction or, conversely, a single node may be replaced by a subnet to show greater detail in the model.

Petri nets, as a formal graph model, are especially useful in modelling the flow of information and control in systems which can be characterized by asynchronous and concurrent behavior. Figure II.B.5 shows the initial marking of a Petri net model of such a system. Initially, transition E1 is enabled because each of its input places, C1 and C2, is marked with a token. Firing transition E1 removes one token each from places C1 and C2, and puts a token into each output place, C3 and C4. At this point in the net execution, transition E3 is disabled because one of its input places, C5, still has no token. Transition E2, however, is enabled, and upon firing causes a token to be removed from place C3 and one deposited in place C5. As an aside, this portion of the model could correspond to a system sequencing

constraint, that of event E3 having to wait until event E2 completes. Upon firing transition E3, places C6 and C7 become marked with tokens as places C4 and C5 lose a token each. Transitions E4 and E5 are now enabled and can fire simultaneously, the occurrence of which corresponds to concurrent events in a modelled system. Doing so, that is, firing transitions E4 and E5, brings the Petri net model back to its original (initial) configuration.

One other situation that can be represented using Petri nets is that of conflict. Figure II.B.6 shows a net model of such a situation. Simply, transitions E1 and E2 are both enabled. However, if either transition fires, the remaining transition becomes disabled. In such a case, it is an arbitrary decision as to which one fires. Because we would like to be able to duplicate experiments and obtain the same results, a scheme that is often used involves simply assigning priorities to transitions which are subject to conflict in a net. In this way reproducible results can be ensured. If true nondeterminism is desired in such a model, a scheme in which probabilities are associated with each transition can effectively model nondeterminism in the system under study.

Thus, to properly model a system with Petri nets, every sequence of events in the modelled system should be possible in the Petri net and every sequence of events in the Petri

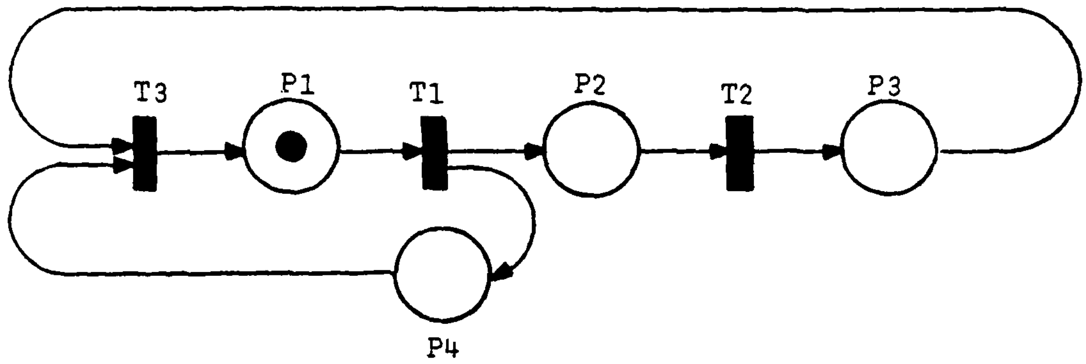


Figure II.B.1: MARKED PETRI NET, TIME=0

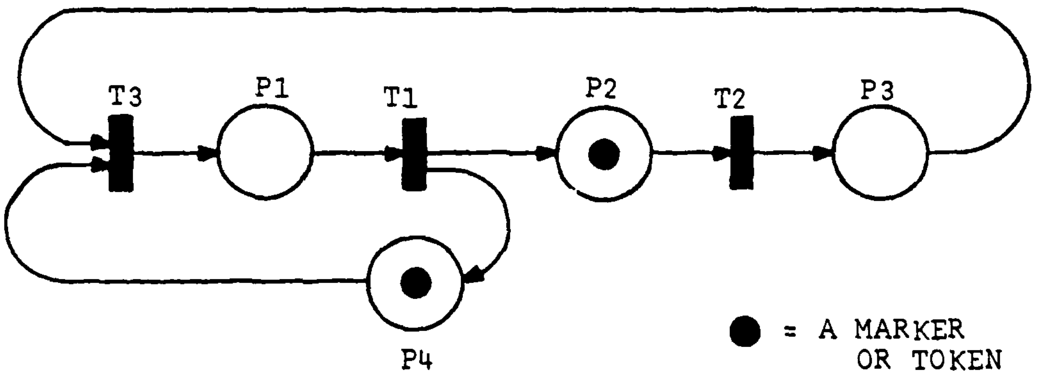


Figure II.B.2: MARKED PETRI NET, TIME=1

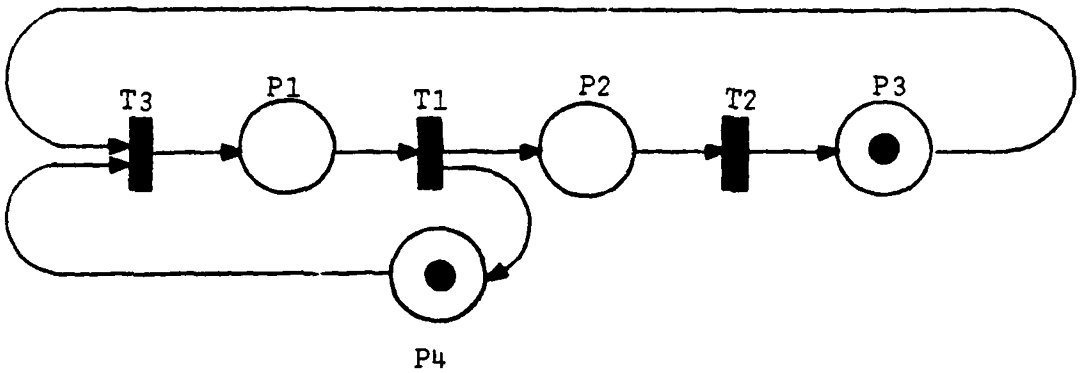


Figure II.B.3: MARKED PETRI NET, TIME=2

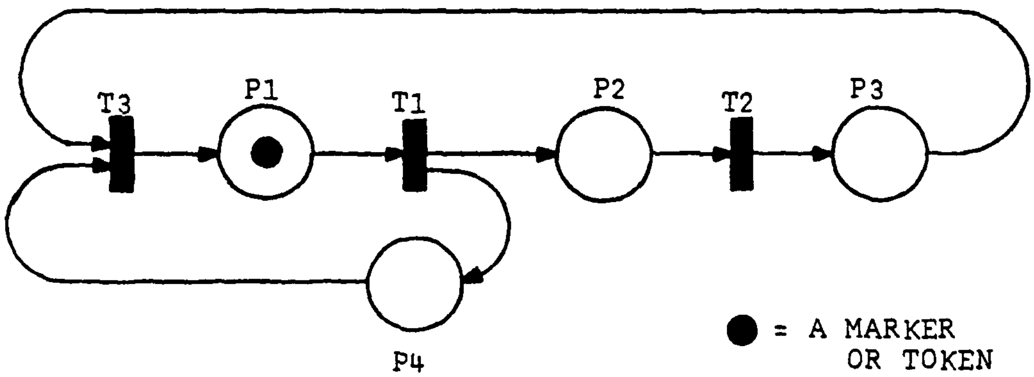


Figure II.B.4: MARKED PETRI NET, TIME=3

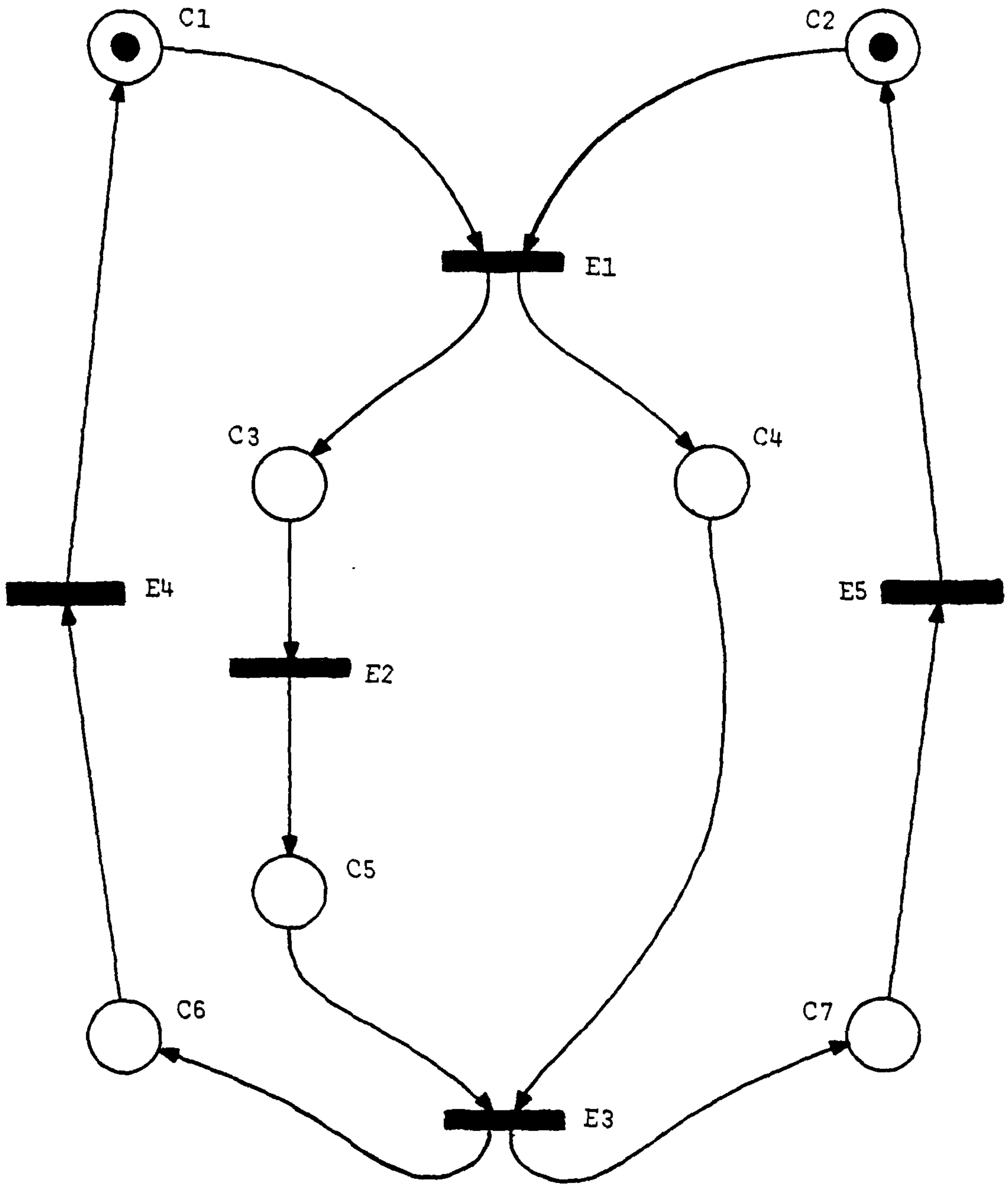


Figure II.B.5: PETRI NET GRAPH MODELLING
ASYNCHRONOUS, CONCURRENT BEHAVIOR

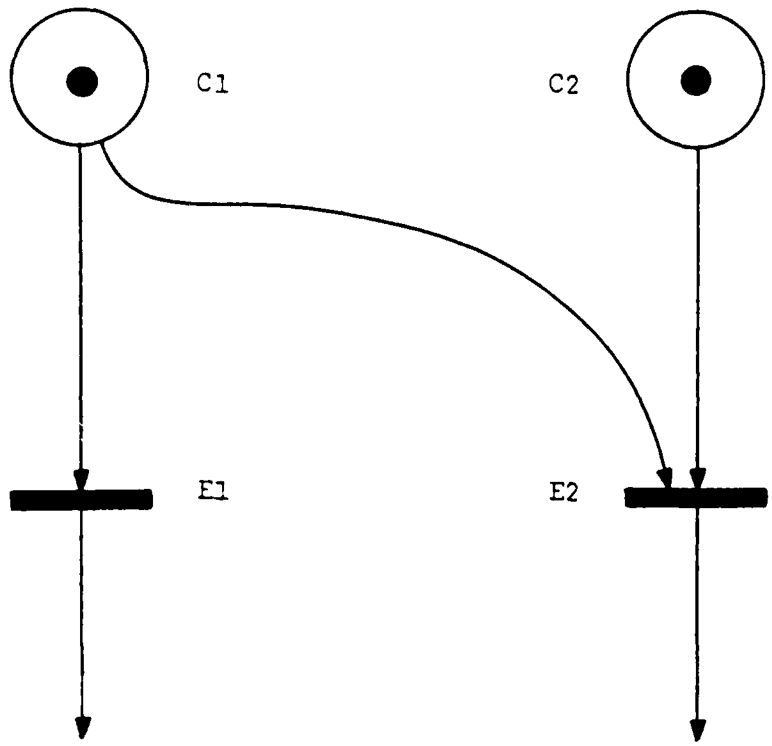


Figure II.B.6: PETRI NET GRAPH MODELLING CONFLICT

net should represent a possible sequence in the modelled system.

This section has introduced Petri nets and demonstrated their usefulness in formally modelling information and control flow in systems characterized by asynchronous and concurrent behavior. Readers interested in the use of Petri nets for performance evaluation of such systems are referred to [RAMAMOORTHY, 1980] and [RAMCHANDANI, 1974]. The following section shall introduce readers to the concept of data flow which, as mentioned previously, is a direct application of Petri net theory.

C. CONCEPT OF DATA FLOW

Data flow computing is a method of multiprocessing which proposes to exploit inherent algorithmic parallelism at all hierarchical levels within a program. Additional objectives include effectively using the capabilities of LSI technology and simplifying the programming task. The concept of computation under data flow was derived by Dennis [DENNIS, 1974] (and a number of others working independently), predominantly from Karp and Miller's [KARP, 1966] work on computation graphs. This section begins by presenting the data flow concept from the perspective of language, rather than that of architecture. This approach is appropriate in view of the fact that data flow computer systems are being designed as hardware interpreters for a base language that

is fundamentally different from conventional languages. A hardware description of the Dennis-Misunas data flow architecture design completes this section.

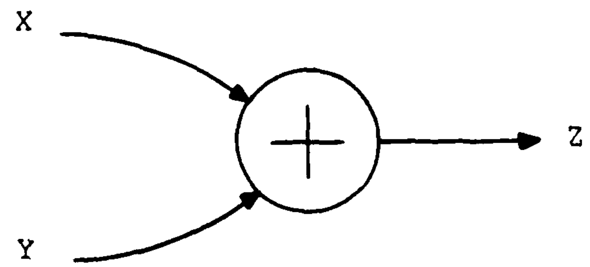
In a data flow computer, an instruction is executed as soon as its operands become available. No notion of separate control flow exists because the data dependencies define the flow of control in a data flow program. In fact, data flow computers have no need for a program location counter.

This contrasts with the traditional "von Neumann" computer architecture model which uses a global memory whose state is altered by the sequential execution of instructions. Such a model is limited by a "bottleneck" between the computer control unit and the global memory [BACKUS, 1978]. This "feature" allows conventional languages to have side-effects, a common example of which is the ability of a procedure to modify variables in the calling program. Such side-effects are prohibited under the data flow concept. Furthermore, in data flow, no variables exist, nor are there any scope or substitution rules. In fact, the data flow concept prohibits the modification of anything that has a value. Rather, data flow computing takes inputs (operands) and generates outputs (results) that have not previously been defined. Thus, instructions in data flow are pure functions. This is necessary so that instruction execution can be based solely on the availability of data (operands). Thus the data dependencies must be equivalent

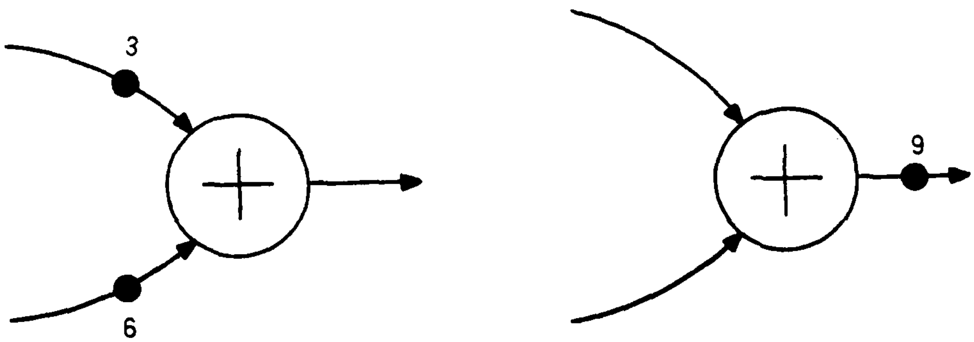
to, and in fact define, the sequencing constraints in a program. Also, to exploit parallelism at all levels, it must be possible to derive these data dependencies from the high level language program instructions [ACKERMAN, 1979].

A language which allows processing by means of operators applied to values is called an applicative language. VAL (Value-oriented Algorithmic Language) is a high level data flow applicative language under development at MIT [ACKERMAN and DENNIS, 1979]. It prevents any side-effects by requiring programmers to write expressions and functions; statements and subroutines are not allowed in the language. Because of this constraint, most concurrency is apparent in a high level language program written in VAL. For the purposes of this research, no further understanding of the high level language of data flow is required. Information about high level language alternatives is available in [McGRAW, 1980], [ACKERMAN and DENNIS, 1979], and [ACKERMAN, 1979].

At what would correspond to the assembly language level, a data flow computation can be represented as a graph. The nodes of the graph correspond to operators and the arcs represent data paths. An arc into a node represents an input operand path; an arc leaving a node corresponds to a result path. Data flow graph execution occurs as operands become available at each node. When the input arcs of a node each have a value on them, the node can execute by removing those values, computing the operation, and placing the results on

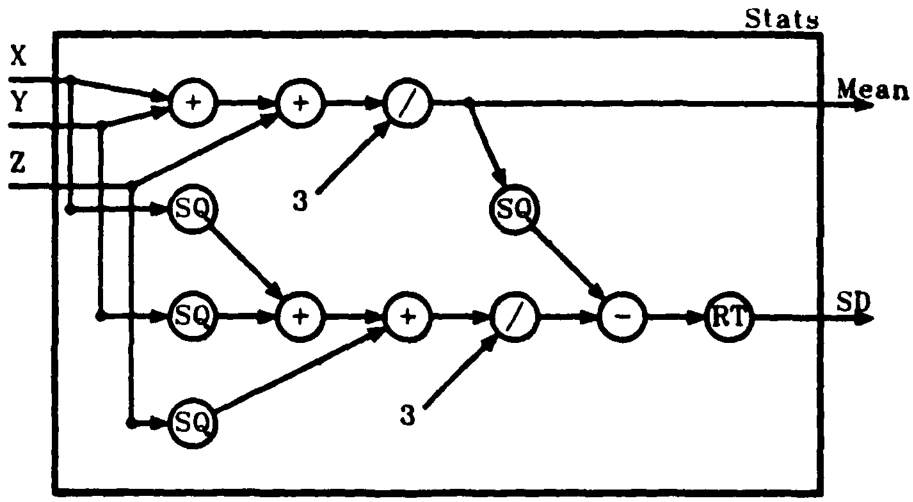


$$Z = X + Y$$



NODE ENABLED　　　　NODE AFTER EXECUTING

Figure II.C.1: DATA FLOW NODE EXECUTION




```
function  Stats (X,Y,Z: real  returns real, real)
   let
       Mean real := (X + Y + Z) / 3;
       SD real := SQRT( (X² + Y² + Z²) / 3 - Mean² );
   in
       Mean , SD
   endlet
   endfun
```

Figure II.C.2: A SIMPLE STATISTICS FUNCTION AND ITS DATA FLOW GRAPH [McGRAW, 1980]

the output arcs (see Figure II.C.1). The example data flow graph in figure II.C.2 computes the mean and standard deviation of its three input parameters.

Such a graph notation is useful in illustrating the various levels of parallelism in a program. For example, a graph node may represent a simple operator such as addition, or the entire statistics function of figure II.C.2. Thus the data flow graph notation can represent parallelism existing at the operator, function and even computation level. The graphs execute asyncnronously, nodes firing when data inputs are available. Thus no syncnronization problem exists with regard to accessing shared data. Each data flow path can be marked with a value by only one operator node. Once a value is on a path, no operator can modify that value. The value can only be read when used as an input to another node [McGRAW, 1980].

Again, the data flow graph notation merely allows a logical representation of a program at a level corresponding to conventional assembly language. This logical representation shall now be extended to permit the reader to understand the basic data flow hardware instruction execution mechanism. A simple example computation that shall be used to facilitate reader understanding is the following:

$$Z = (X + Y) * (X - Y)$$

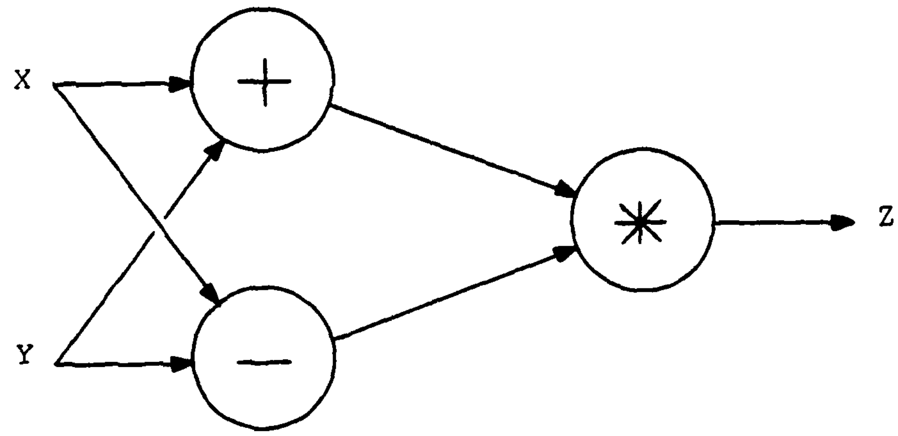


Figure II.C.3: A SIMPLE DATA FLOW PROGRAM GRAPH

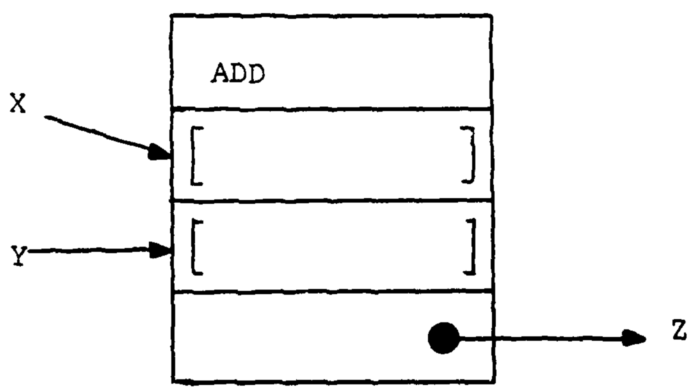


Figure II.C.4: AN ACTIVITY TEMPLATE FOR THE ADDITION OPERATOR

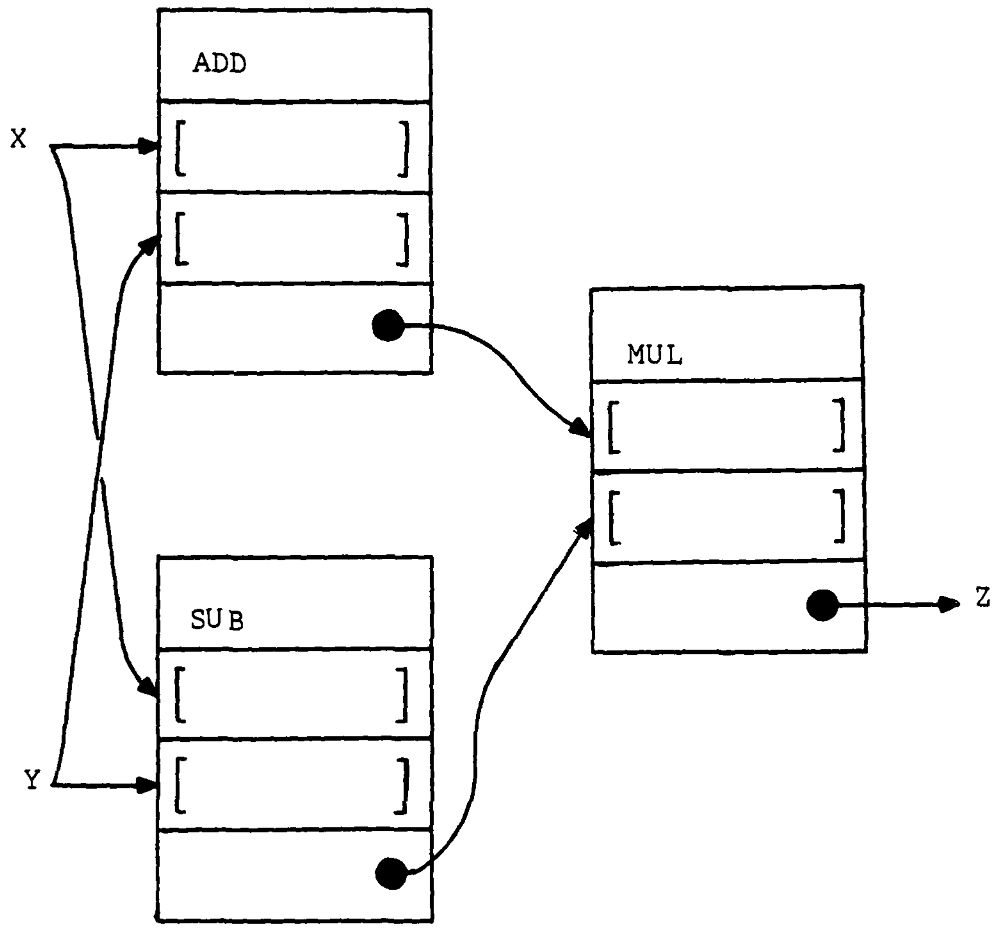


Figure II.C.5: PROGRAM GRAPH USING ACTIVITY TEMPLATES FOR THE DATA FLOW PROGRAM GRAPH OF Figure II.C.3 [DENNIS,1980]

The graph representation of this computation is shown in figure II.C.3.

In the extended graph representation scheme, a data flow program exists as a collection of activity templates, each template corresponding to a node in the data flow program graph. For example, figure II.C.4 shows an activity template for the addition operator. There are four fields in the activity template. The first field denotes the operation code which specifies the operation to be performed. The second and third fields are receivers, which are locations waiting to receive operand values. The fourth field is a destination field which specifies where the result of the operation on the operands is to go. There can be multiple destination fields. Figure II.C.5 shows the program graph representation of figure II.C.3, using activity templates.

Activity templates have been developed which control the routing of data for such program structures as conditionals and iterations. These templates are mentioned to point out the fact that graph nodes can represent not only simple operands but can also represent more elegant and necessary constructs.

Some definitions which are necessary to the understanding of the data flow instruction execution mechanism follow. First, a data flow program instruction is the fixed portion of an activity template and is made up of the opcode and the destinations.

instruction:

<opcode, destinations>

Each destination field provides the address of some activity template and an input (or offset) denoting which receiver of the template is the target.

destination:

<address, input>

Data flow program execution occurs as follows. The fields of a template which has been activated (by the arrival of an operand value at each receiver) form an

operation packet:

<opcode, operands, destinations>

When the operation packet has been operated upon, a result packet of the form

result packet:

<value, destination>

is generated for each destination field of the original activity template. Result packet generation triggers the placement of the value in the receiver designated by its destination field. Thus, at a logical level, data flow program execution occurs as a consequence of operation

packet and result packet movements through a machine described in detail below.

The basic data flow instruction execution mechanism is shown in figure II.C.6. The data flow program, consisting of a collection of activity templates, is held in the activity store (see figure II.C.6). Each activity template is uniquely addressable within the activity store. When an instruction is ready to be executed (i. e. the template is enabled), this address is entered in the instruction queue unit (established as a FIFO buffer).

The fetch unit is then responsible for: removing, one at a time, instruction addresses from the instruction queue, fetching the corresponding activity template, forming an operation packet based on the field values in the template, and submitting the operation packet to an operation unit for processing. The operation unit processes the operation packet by: performing the operation specified by the opcode on the operands, forming result packets (one for each destination field of the operation packet), and transmitting the result packets to the update unit. The update unit fills in the receivers of activity templates (designated by the destination fields in the result packets) with the appropriate values. The update unit is also responsible for checking the target template to see if it has all receivers filled, thus enabling the template. If so, the address of

the enabled template is added at the end of the instruction queue by the update unit.

At this point it is appropriate to discuss how and where program parallelism can be exploited by this hardware.

"...once the fetch unit has sent an operation packet off to the operation unit, it may immediately read another entry from the instruction queue without waiting for the instruction previously fetched to be completely processed. Thus a continuous stream of operation packets may flow from the fetch unit to the operation unit so long as the instruction queue is not empty.

"This mechanism is aptly called a circular pipeline-activity controlled by the flow of information packets traverses the ring of units leftwise. A number of packets may be flowing simultaneously in different parts of the ring on behalf of different instructions in concurrent execution. Thus the ring operates as a pipeline system with all of its units actively processing packets at once. The degree of concurrency possible is limited by the number of units on the ring and the degree of pipelining within each unit. Additional concurrency may be exploited by splitting any unit in the ring into several units which can be allocated to concurrent activities." [DENNIS, NOV1980]

The Dennis-Misunas data flow architecture for implementing the described instruction execution mechanism is called the cell block architecture and is illustrated in figure II.C.7.

"The heart of this architecture is a large set of instruction cells, each of which holds one activity template of a data flow program. Result packets arrive at instruction cells from the distribution network. Each instruction cell sends an operation packet to the arbitration network when all operands and signals have been received. The function of the operation section is to execute instructions and to forward result packets to target instructions by way of the distribution network." [DENNIS, NOV1980]

Figure II.C.8 reflects a practical form of the cell block architecture which makes use of LSI technology and reduces the number of devices and interconnections. This practical form is obtainable by grouping the instruction cells of figure II.C.7 into blocks, each of which is a single device. In this organization, several cell blocks are serviced by a group of multifunction processing elements. The arbitration network channels operation packets from cell blocks to processing elements. Rather than employing a set of processing elements each capable of a different function, which is one design option, use of one multipurpose processing element type is the favored approach. Such an approach precludes the need for the arbitration network to route operation packets according to opcode. Instead, it simply has to forward operation packets to any available processing element. It is this design which forms the basis for the system model used in this research effort.

How does the basic mechanism relate to the cell block architecture? Figure II.C.9 shows a cell block implementation. It differs from the basic mechanism in two ways. First, the cell block has no processing element(s) (operation unit(s)). Second, result packets targeted for activity templates held in the same cell block must traverse the distribution network before being handled by the update unit [DENNIS, NOV 1980]. This is the Dennis-Misunas data

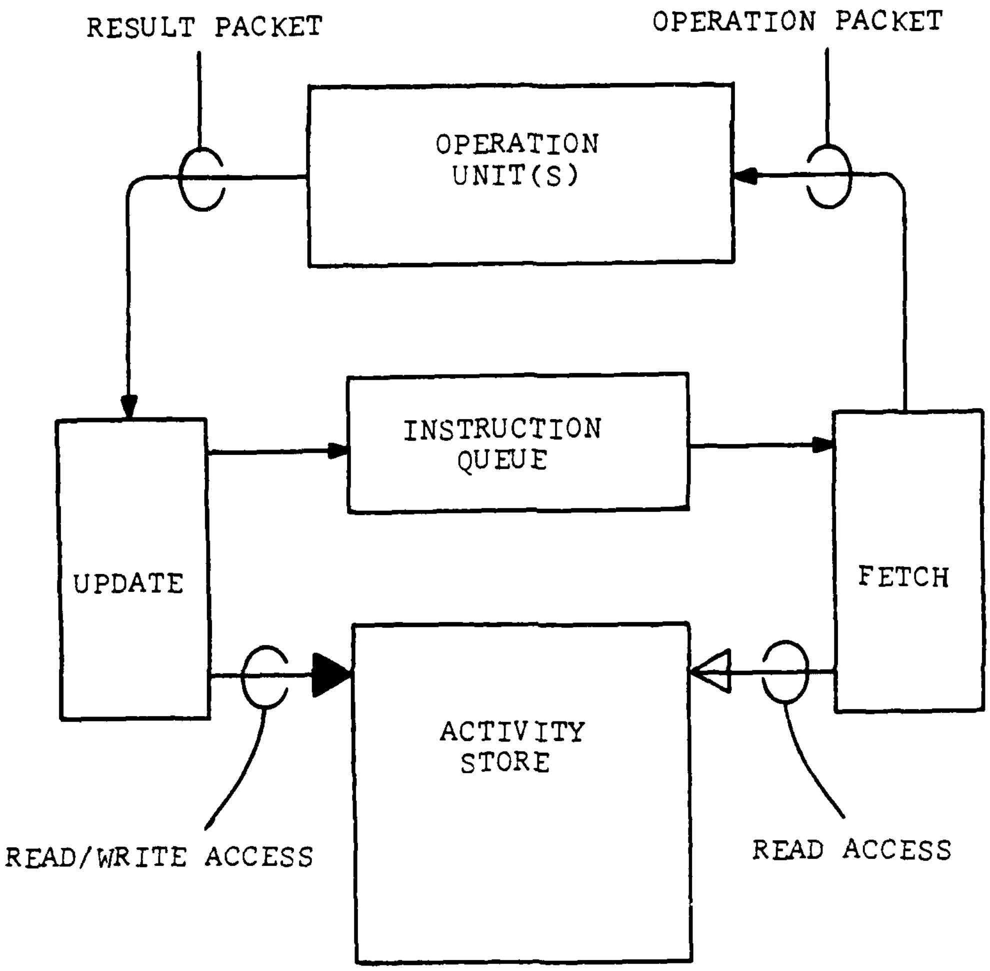


Figure II.C.6: DATA FLOW INSTRUCTION EXECUTION MECHANISM [DENNIS, 1980]

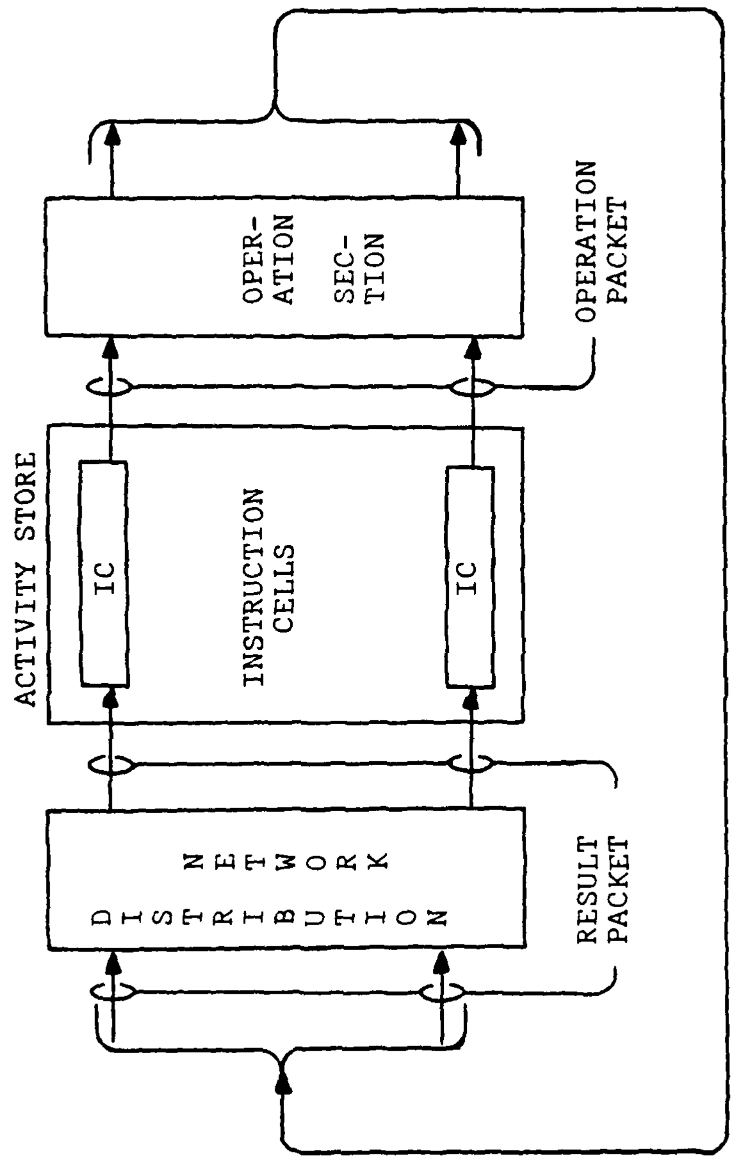


Figure II.C.7: DENNIS-MISUNAS CELL BLOCK ARCHITECTURE [DENNIS, 1980]

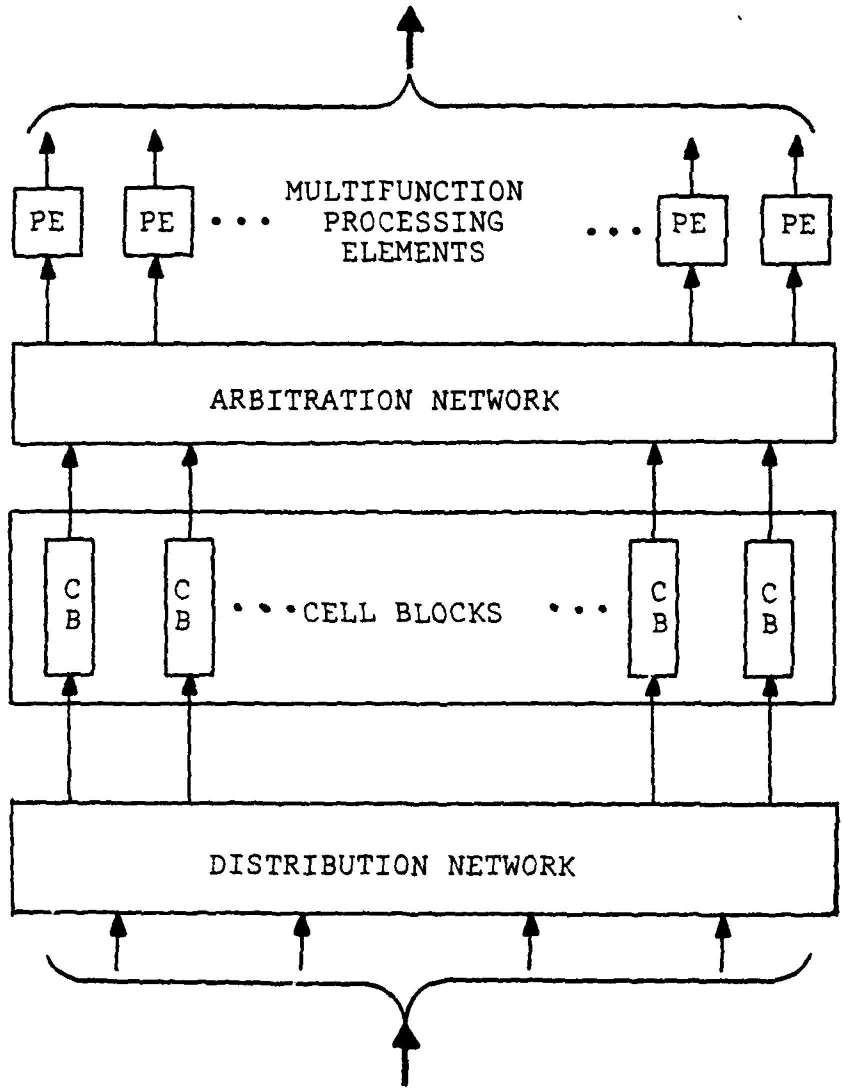


Figure II.C.8: PRACTICAL FORM OF THE CELL BLOCK ARCHITECTURE [DENNIS, 1980]

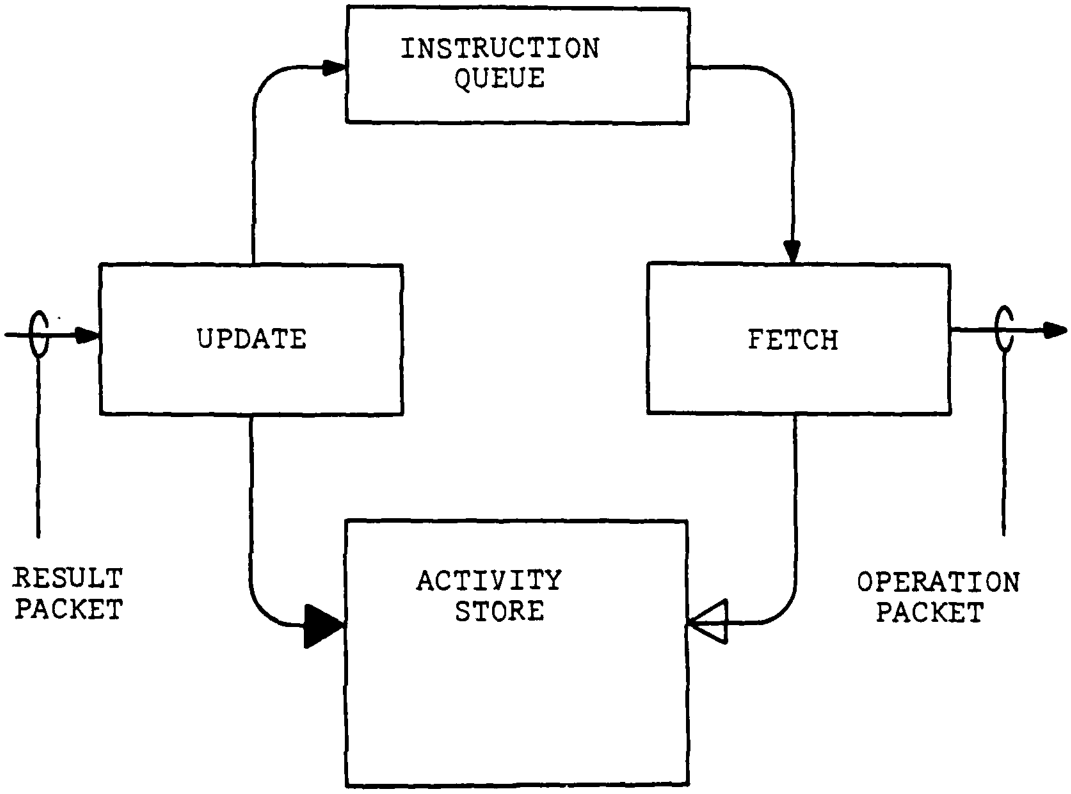


Figure II.C.9: A SIMPLE CELL BLOCK IMPLEMENTATION

[DENNIS, 1980]

flow architecture design. Other designs do exist; for examples, see [GOSTELOW, 1980] and [WATSON, 1979].

D. COMPUTER PERFORMANCE PREDICTION

Computer performance prediction is an evaluation process which proposes to estimate the performance of a system not yet in existence (i.e. in some state of design). "Performance" simply means how well a system works. This in turn connotes the concept of value. So, the purpose of estimating the performance of a system under design is to determine that system's expected value.

In order to quantify how well a system works or shall work, performance metrics called indices are used. Typical indices and their definitions are:

THROUGHPUT RATE - The volume of information processed by a system in one unit of time

HARDWARE UTILIZATION - The ratio between the time the hardware is used during an interval of time, and the duration of that interval of time

RESPONSE TIME - The elapsed time between the submission of a program job to a system and completion of the corresponding job output.

Computer performance prediction can be achieved via several different techniques. Each technique has limitations and advantages. The technique utilized in this thesis is that of simulation. The simulation technique involves the representation, by a model, of certain aspects of the behavior of a system in the time domain. Observing these

aspects of the behavior in time of the system's model, under inputs generated by a model of the system's inputs, produces results useful in the evaluation of the modelled system [FERRARI, 1978]. For the purposes of this research, the aspects of behavior that are of interest are the performance indices previously defined.

Of significant importance to any simulation effort are the issues of validation and parameter estimation. Conceptually, validation attempts to establish some degree of confidence that the simulation shall produce results which shall closely correspond with the performance of the system under scrutiny. Parameter estimation provides the simulation effort with hopefully credible parameter values needed to perform a simulation having relevant results. These issues shall be addressed in section IV.A: Experimental Design.

The last section of the review of the literature applicable to this research endeavor presents the Requester-Server methodology. The Requester-Server methodology is the "tool" used to perform the simulation which generates the results on which the prediction of data flow performance is based.

Readers desiring a more thorough presentation of the subject of computer performance prediction are referred to [FERRARI, 1978], [COX, 1978], [ALLEN, 1980], [HAMMING, 1975], [SPRAGINS, 1980], [BUZEN, 1980], and [SAUER, 1980].

E. REQUESTER-SERVER METHODOLOGY

The Requester-Server (R-S) methodology was designed and initially implemented by L. A. Cox, Jr. [COX, 1978]. Subsequently, the Requester-Server software was modified by D. M. Stowers [STOWERS, 1979] to run on the PDP-11/50 minicomputer at NPS. This section summarizes those portions of [COX, 1978] and [STOWERS, 1979] which are applicable to and necessary for the understanding of this research.

The R-S methodology is capable of predicting the performance of computer systems characterized by asynchronous, concurrent behavior. The methodology can predict performance at both the computer system and computer job levels. The R-S methodology allows the user to separately specify the hardware configuration(s) to be evaluated, the software (programs) to be used in evaluating the hardware configuration(s), and the mechanism or policy for allocating hardware resources to program requests for service. The methodology makes provision for variable levels of detail (in a hierarchical sense) in both the hardware and software. Finally, the R-S methodology is capable of simulating concurrency in both the hardware and software. Thus, for a given hardware configuration, the control structure mandated by the software can be mapped onto the hardware and system performance analyzed and predicted.

The simulation process is begun by representing the software (programs) and hardware as two separate Petri net

graphs. In the Petri net graph of the software, each arc can be thought of as having an associated propagation delay, the extent of which is dependent upon the hardware configuration used to execute the program. If these delays are definable by their correlation to the Petri net model of the hardware, then performance values for the indices of section II.D (Computer Performance Prediction) can be obtained by executing the Petri net model of the software on the Petri net hardware configuration(s). The R-S "tool" serves as the interface between the Petri net model of system software and Petri net model of system hardware. This interface permits the hardware and software Petri net graphs to be constructed separately. This is important because the control structure and sequencing constraints of both hardware and software can be maintained separately. This permits a direct and meaningful representation of both the system software and hardware being modelled.

The source file which serves as the input to the R-S program is organized into three sections. The software section of the input file consists of a description of the Petri net graph representing the software program(s) to be executed. This net graph description is formulated in terms of the functions and constraints of the services required of the hardware. The hardware section of the input file is made up of a description of the computer system components and their interconnections. This description can be

(hierarchically) at a bit-level or major component level, depending on the system aspects under scrutiny. The Petri net graph upon which the hardware description is based is constructed in terms of its operation in time. The last section of the input file, called the dynamic section, provides the user of the R-S "tool" a place to denote system initial conditions by defining the hardware and software nets' token markings at the beginning of a "run". As may be recalled from section II.B (Petri Nets), both the software and hardware sections merely define static Petri net structures. Performance prediction follows from the attachment of significance to the structures and restrictions on token movement within these structures.

The dynamic nature of Petri nets is exploited by this R-S methodology as follows. The software net representation makes a series of requests for the services of the hardware net representation. Repeatedly, the R-S process maps these requests for service onto the hardware net representation. At each "invocation" the R-S process "runs" the hardware net to provide the service requested by the software net. Upon completion of each of the service requests, the R-S process "runs" the software net representation until the hardware is again needed. This cycle repeats itself until the software net representation has been completely "run" and its terminal state reached.

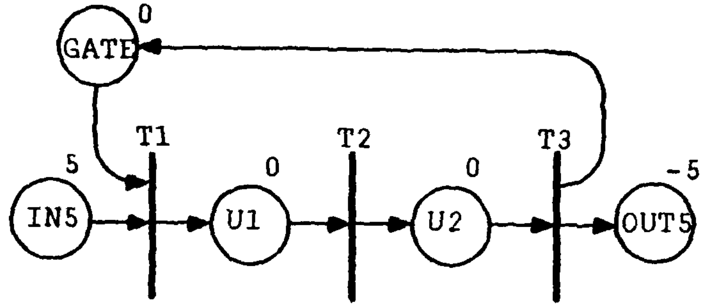
Events in the hardware net graph correspond to operations in time. A collection of events is used to represent each functional unit. Token movement through the hardware net graph corresponds to the flow of data and control through the modelled hardware system. A simple hardware net description is provided in figure II.E.1. Events in the software net graph correspond to requests for service. As an example, an event could equate to a request for a floating point multiplication. The flow of tokens in the software net graph equates to the logical flow of the algorithm, constrained by its implicit data dependencies or sequencing constraints. A simple software net description is provided in figure II.E.2.

Together, the software and hardware net graphs can be executed in such a way as to simulate the operation of the computer system for the given software workload. The interaction of the two net graphs is orchestrated by the R-S token arbiter. Network simulation begins with the marking of the "BEGIN" node of the software net graph. This net graph is then executed as would be any Petri net graph. The arrival of a token at any place in the software net graph indicates a request for service, at which time the R-S token arbiter takes control. (The type of service requested is denoted by the type of the place and is defined in the software net description.) The R-S token arbiter removes the token from the software net and then permits the software

net graph to continue executing until no further moves are possible. The R-S token arbiter then initializes the hardware functional unit (net graph denoted by the type of service requested) by marking it with tokens. The hardware net graph is then executed one step. Tokens reaching events corresponding to service completion are removed, and the token of the software net which originally caused the request for service is replaced, by the R-S token arbiter. Repeating this sequence of actions results in the execution of the software net graph by the hardware net graph. A sample input file dynamic section and the results obtained from executing the software and hardware net graph descriptions of figures II.E.1 and II.E.2 are presented in figure II.E.3. Those readers interested in the Requester-Server methodology are referred to [COX, 1978] and [STOWERS, 1979] for a more in-depth discussion of its capabilities and usage.

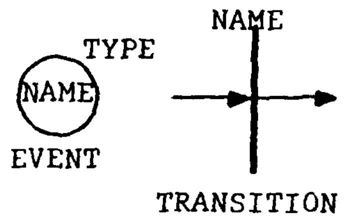
This completes the necessary review of the literature required to understand the research that follows. The fundamental concepts of the various approaches to parallelism, Petri nets, the data flow architecture, computer performance prediction, and the Requester-Server methodology have been reviewed. The next section presents the two-part hypothesis which this research addresses and tests.

```
BEGIN MACHINE NET;
DECLARE GATE EVENT TYPE 0;
DECLARE IN5 EVENT TYPE 5;
DECLARE U1 EVENT TYPE 0;
DECLARE U2 EVENT TYPE 0;
DECLARE OUT5 EVENT TYPE -5;
DECLARE T1 TRANSITION;
  INPUT IN5;
  INPUT GATE;
  OUTPUT U1;
END T1;
DECLARE T2 TRANSITION;
  INPUT U1;
  OUTPUT U2;
END T2;
DECLARE T3 TRANSITION;
  INPUT U2;
  OUTPUT GATE;
  OUTPUT OUT5;
END T3;
END MACHINE NET;
```



AN ADDER (3 MINOR CYCLES)
NOT PIPELINED

LEGEND:



(TYPE refers to kind of service provided or requested)

Figure II.E.1: A SAMPLE HARDWARE NET GRAPH AND INPUT FILE DESCRIPTION

```
BEGIN PROGRAM EXAMPLE;
DECLARE BEGIN EVENT TYPE 0;
DECLARE J+K EVENT TYPE 5;
DECLARE M+J EVENT TYPE 5;
DECLARE END EVENT TYPE 0;
DECLARE ST1 TRANSITION;
  INPUT BEGIN;
  OUTPUT J+K;
  OUTPUT M+J;
END ST1;
DECLARE ST2 TRANSITION;
  INPUT J+K;
  INPUT M+J;
  OUTPUT END;
END ST2;
END PROGRAM EXAMPLE;
```

FORTRAN PROGRAM:

```
C   BEGIN (EVERYTHING IN REGISTERS)
    I = J + K
    L = M + J
C   END
```

PETRI NET REPRESENTATION:

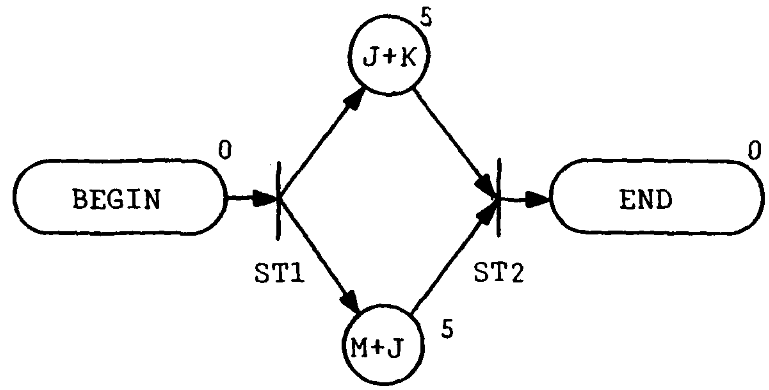


Figure II.E.2: A SAMPLE SOFTWARE NET GRAPH AND INPUT FILE DESCRIPTION

```
BEGIN DYNAMIC NET;

MARK GATE WITH 1;
COMMENT:  GATE ENABLED TO ALLOW
          ONLY ONE OPERATION IN
          PROGRESS AT ANY TIME.

EXECUTE 10;
COMMENT:  EXECUTE TEN HW CYCLES
          OR  UNTIL  PROGRAM IS
          COMPLETE.

END DYNAMIC NET;
```

```
S5:  EXECUTE 10;
     *PROGRAM EVENT J+K REQUESTS HW SVCS(1)
     *PROGRAM EVENT M+J REQUESTS HW SVCS(1)

TIME = 1:
TIME = 2:
TIME = 3:
        *PROGRAM EVENT J+K COMPLETES(3)
TIME = 4:
TIME = 5:
TIME = 6:
        *PROGRAM EVENT M+J COMPLETES(6)

S6:  END DYNAMIC NET;
```

Figure II.E.3: A SAMPLE INPUT FILE DYNAMIC SECTION AND OUTPUT FILE LISTING

III. HYPOTHESIS

Because their exist several data flow architecture proposals, it is desirable to have a tool with which to predict the performance of the diverse designs for comparison purposes. The first part of this research's hypothesis was that the Petri net-based Requester-Server (R-S) methodology is such a tool, capable of predicting the performance of data flow architectures in an efficient, accurate manner. In effect, the R-S tool was to be tested.

The second part of this research's hypothesis was concerned with the Dennis-Misunas data flow architecture design. This design was chosen for two reasons. First, there existed adequate information in the literature about this design on which to base an accurate model for simulation purposes. Second, the Dennis-Misunas design of the basic instruction execution mechanism is essentially the same as several other schemes in various stages of implementation [DENNIS, 1979]. The hypothetical challenge to this design was that the goal of achieving higher speed computation is not attainable unless a high and "intelligent" degree of multiprogramming is realized, as shall be explained next.

Obviously, high speed computation shall require a high hardware utilization. By this it is meant that most of the processing elements (PE's) shall have to be performing

useful work most of the time. Such a high hardware utilization is attainable when either of two situations occurs. First, a high hardware utilization will result when a process possessing a large amount of inherent parallelism is being run (by itself) on the machine. In this case, a program's execution time is dependent upon its amount of inherent parallelism and the number of PE's in the machine. Second, a high hardware utilization is attainable when a multiprogramming environment (in which several processes are permitted to simultaneously run on the machine) is instituted. In such a multiprogramming environment, an individual process shall be competing for hardware resources (PE's). Thus, that process' execution time may be lengthy regardless of its amount of inherent parallelism. This is so because that process may have the use of only a small portion of the machine's resources (PE's) at any point in time. Put another way, if at any time a process has N instructions available for execution, but there are less than N PE's available for executing those instructions in a parallel fashion, then the process' execution time shall be lengthened over what it could be if it had sufficient PE's available.

In such a situation, a scheme may be needed to implement a policy which achieves two objectives:

1. maintaining high hardware utilization and
2. providing an acceptable average response time for a user requiring a given amount of processing.

"Acceptable average response time" is construed to mean that the actual response time of any particular program which requires a given amount of processing shall not be lengthened considerably over what it would be if the program were executed by itself on the data flow machine. Thus, it shall be desirable to minimize the affect of system load on an individual program's execution time. That the second objective should be met even at the expense of the first objective is a strong point made by [KLEINROCK, 1976]. By merely mapping processes onto the data flow machine as they arrive, it is expected that objective #1 shall be achieved but at the expense of objective #2. This situation was expected to be demonstrated by this research.

The purpose of this section has been to "frame" the research area by presenting the issues which give rise to the hypothesis. The following section presents the method used to test the hypothesis, and includes a discussion of the assumptions made to facilitate the simulation experiments, where undecided design issues remain.

IV. METHOD

A. EXPERIMENTAL DESIGN

The experiment to allow prediction of data flow computer performance involved executing sets of Petri net models of data flow programs on Petri net models of data flow hardware. The Requester-Server (R-S) program tool monitored the data flow (model) programs' "execution" and provided data which permitted the determination of the performance indices: response time and hardware utilization. It is important to realize that the results of this research predict the performance of a model of a data flow computer-not that of an operating data flow machine itself. (Model validation and parameter estimation issues are addressed in section IV.B: Data Flow Hardware Definition.)

The reader who is familiar with analytic modelling employing queueing theory may ask why that technique, rather than the simulation technique, is not used to predict the performance of the data flow design. The answer is that the analytic approach unnecessarily constrains the prediction by requiring assumptions to be made about the software. Specifically, the Petri net models of software programs are discretely defined with regard to the amount of inherent parallelism available for exploitation at each time step in program execution. To model analytically, the variability of

inherent parallelism available for exploitation must be described by probability distributions which hide the definable nature of the programs at discrete time steps.

For this experiment, the data flow architecture was to be modelled with several different quantities of processing elements (PE's). The sample data flow program models were to be characterized by varying but definable amounts of inherent parallelism available for exploitation. Each (model) program was to be separately run on each (model) hardware configuration. (Hereafter, the word "model" shall be omitted but assumed in referring to the program and hardware models used in this experiment.) Data was to be obtained to permit determination of the performance indices (response time and hardware utilization), for each run, from the monitor function of the R-S tool. After running each program separately, arbitrary program mixes were to be run on each hardware configuration and the same performance indices again determined. Finally, hand-optimized program mixes were to be run on each hardware configuration and the same performance indices determined once again. By evaluating the results, the hypothesis was expected to be either supported or refuted.

The independent variable for this experiment was defined to be the quantity of PE's available to the hardware model. Because PE's are but one resource demanded by a process in execution, other independent variable choices could have

included other resources such as: the quantity of cell blocks available, the type of distribution or arbitration network employed, and/or the type of PE's (multipurpose or sets of single-purpose functional units) utilized. Expanding the number of independent variables increases significantly the complexity of evaluating the results. How these issues were resolved is explained in section IV.B.

The dependent variables for this experiment were the parameters response time and hardware utilization. The results of the experiment were expected to provide data which could be plotted on graphs. Curves plotting the execution time of each data flow program against the number of PE's would constitute one such graph. Others, and their significance, are presented in section V: Results and Discussion.

B. DATA FLOW HARDWARE DEFINITION

The Petri net models of the data flow hardware configurations were quantified in terms of their operation in time. Such quantification required assigning time duration values to each portion of the cell block architecture model in such a way as to closely model the hardware. Doing so required several assumptions to be made. Those assumptions shall be addressed individually so as to help substantiate the credibility of the resultant hardware models.

To begin, the processing elements (PE's) were assumed to be multifunctional, capable of executing any instruction routed to it in one "standard" instruction execution time unit. Allowing the PE's to be multifunctional and characterized by a singular execution time simplifies the modelling process.

There were at least two other possibilities that could be accommodated by expansion of the Petri net models. First, each multifunction PE could be replaced by a set of single-purpose PE's, each single-purpose PE defined in terms of its particular instruction execution time, and capable of executing concurrently with other PE's of the set. Second, each PE could be replaced by a subnet in which only one instruction could be executed in any given time step, but the model would define the execution time as a function of the instruction type.

The first alternate approach implies a more complex arbitration network with a conceivably longer routing time. The second alternative would require additional net complexity. (However, this approach would be a good possibility for subsequent research.) Because the actual implementation configuration has not been finalized, modelling the PE's as multifunctional and characterized by a singular execution time was a reasonable path to follow.

The distribution network design also has not been finallized. For ease of modelling purposes, a crossbar

switch design capable of supporting simultaneous transfers of result packets to cell blocks was chosen to be modelled. This choice permitted a standard routing time to be characterized by the model. Other network designs, especially packet routing networks, may be preferred to the crossbar switch for the ultimate machine because of their lower cost and comparable performance in a data flow architecture [DENNIS, 1979].

The choice to model the PE's as multifunction units precluded the need for anything but a simple arbitration network. Such a network would merely have to route operation packets to any available PE. Accordingly, in the model, a standard routing time for this network was characterized.

With regard to the cell blocks, the assumption was made that sufficient cell blocks were available to hold all portions of all processes being run on the machine at each and every instant. Thus, there is no notion of paging portions of processes into and out of memory (the activity store in the case of the data flow architecture). This assumption carries with it the assumption that all program compilation (resulting in extended data flow graph-like representations) is complete before beginning program execution. Other compilation strategies are under consideration, such as requiring the user to interact with the system to achieve a high degree of parallelism exploitation [McGRAW, 1980].

As has been described, other hardware choices (representing independent variables in an experiment) can be made and easily implemented by simply defining appropriate subnets which, in a time-wise fashion, characterize the portions of the hardware under scrutiny. The approach taken in this research permitted the hardware timing characteristics to be a function of simply the number of PE's. Figure IV.B.1 is the Petri net representation (of the cell block architecture hardware) utilized in this research. For the purposes of this experiment and in the configuration described, each PE was assumed to be driven at the rate of two million floating point operations per second (FLOPs), a rate claimed to be reasonable by [DENNIS, 1980]. This figure represents an instruction execution time of 500 nanoseconds (nsec). (This is represented in the hardware model by signifying a scaling of each event/transition pair to equal 100 nsec.) Associating timing characteristics with each component in the data flow architecture design results in a similar figure as shown in figure IV.B.2.

PE (instruction execution)	50 nsec
CELL BLOCK (memory fetch assuming MOS technology)	250 nsec
DISTRIBUTION NETWORK (assuming crossbar switch)	250 nsec
ARBITRATION NETWORK (assuming negligible)	-
[WEITZMAN, 1980].	TOTAL: 550 nsec

FIGURE IV.B.2: TIMING CHARACTERISTICS OF DATA FLOW ARCHITECTURE COMPONENTS

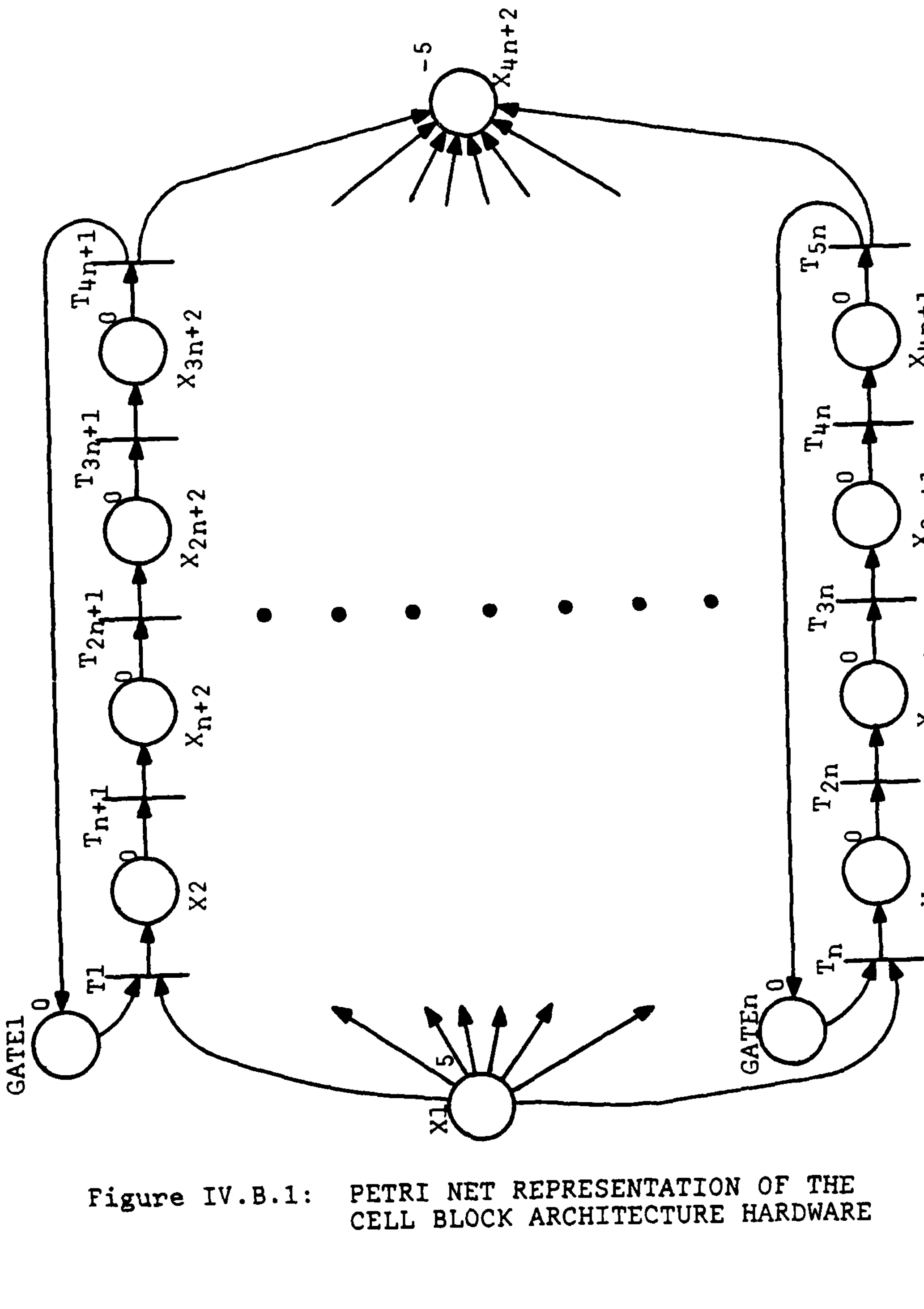


Figure IV.B.1: PETRI NET REPRESENTATION OF THE CELL BLOCK ARCHITECTURE HARDWARE

For the purposes of this research, the quantity of PE's in the hardware was varied from one to sixteen, by multiples of two. This resulted in data generated for the following quantities of modelled PE's: 1, 2, 4, 8, and 16.

This summarizes the assumptions utilized in developing the model presented here. The following section describes the Petri net definition of the data flow software- program models which were "executed" on the hardware models.

C. DATA FLOW SOFTWARE DEFINITION

The Petri net models of data flow programs were quantified in terms of the amount of inherent parallelism available for exploitation at each discrete time step as well as in terms of the implicit data dependencies of the programs. (As previously mentioned, the data dependencies define the control flow of a program.) The initial approach involved taking sample programs written in the high level language (hll) VAL and converting them to their equivalent Petri net representations for subsequent "execution" on the data flow hardware models. The problem with this approach was that the compilation process is not yet developed. Thus, what hardware instructions would be required for each hll instruction were not determinable.

The subsequent approach, which was utilized, involved designing Petri net program models characterized by various but discretely definable levels of inherent parallelism.

Executing such artificial programs conceivably produced more informative results than would have been obtained with a few select programs which may have only demonstrated data flow's suitability for those special purpose computations. The individual programs shall be characterized after introducing a new concept.

A new concept introduced at this point is that of a software "concurrency vector". A concurrency vector is a tuple, each entry of which defines the amount of inherent parallelism in a program at the operation packet hierarchical level, at a discrete instruction execution time step. Each entry of the tuple is implicitly subscripted by the time step it describes. For example, the simple statistics function of section II.C (see figure II.C.2, page 32) would be characterized by the concurrency vector: (4,2,2,2,1,1). In this example concurrency vector, the "4" represents the fact that the four operations "+", "SQ", "SQ", and "SQ" could be processed in parallel during the first time step of execution of the simple statistics function. This is so because no sequencing constraints exist among these four operations. Thus the concurrency vector defines how many operation packets could be parallel processed if all the instructions (i.e. functions- addition, subtraction, division, square, square root) were implemented in hardware. (If they were not, the subfunctional operation packets required by the instruction would be considered in

defining the concurrency vector entries.) It should also be recognized that the concurrency vector, though a function of a program, is dependent upon a standard instruction execution time duration. If the hardware is implemented such that execution time is a function of the instruction type, then the concurrency vector entries could be described at an even lower level than the operation packet level. Such a level would correspond to a basic hardware cycle time, where executing an instruction would require some number greater than one hardware cycles to complete. This additional complexity need not be considered in this research in view of the hardware design approach taken, but could be accommodated by the R-S methodology used here.

Four programs ("A" through "D") were utilized in this research. These programs are differentiable by their length as well as by the amount of inherent parallelism available for exploitation at each time step. The Petri net representations of these programs are shown in figures IV.C.1 through IV.C.4. Additionally, the concurrency vector for each is shown. The program mixes for this experiment included one of each of the three programs, "A", "B" and "D". Another program mix including one of each of the four programs, "A" through "D", was also used.

The following section presents the procedure utilized in executing the experiment. Additionally, the method of

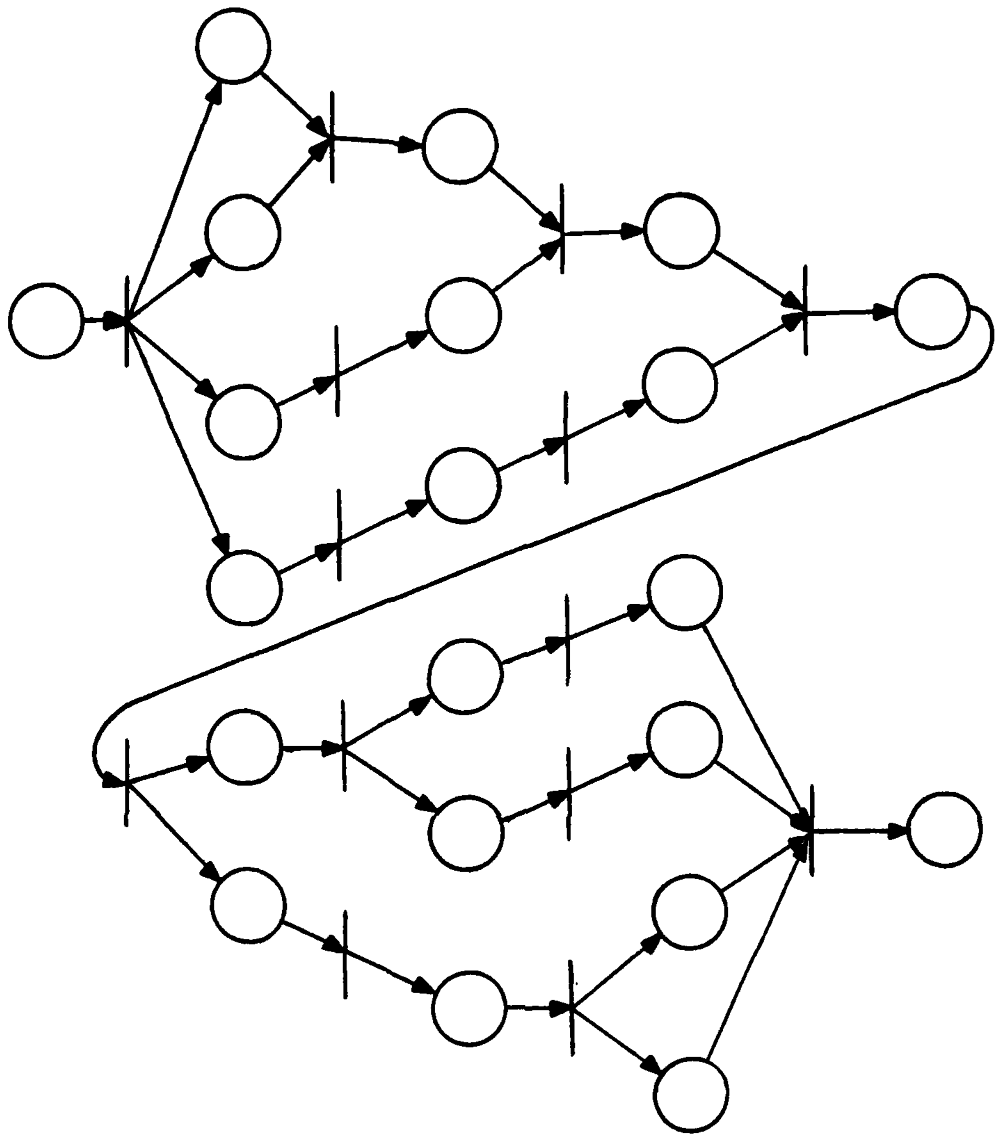


FIGURE IV.C.1: (MODEL) PROGRAM "A"
CV: (4,3,2,1,2,3,4)

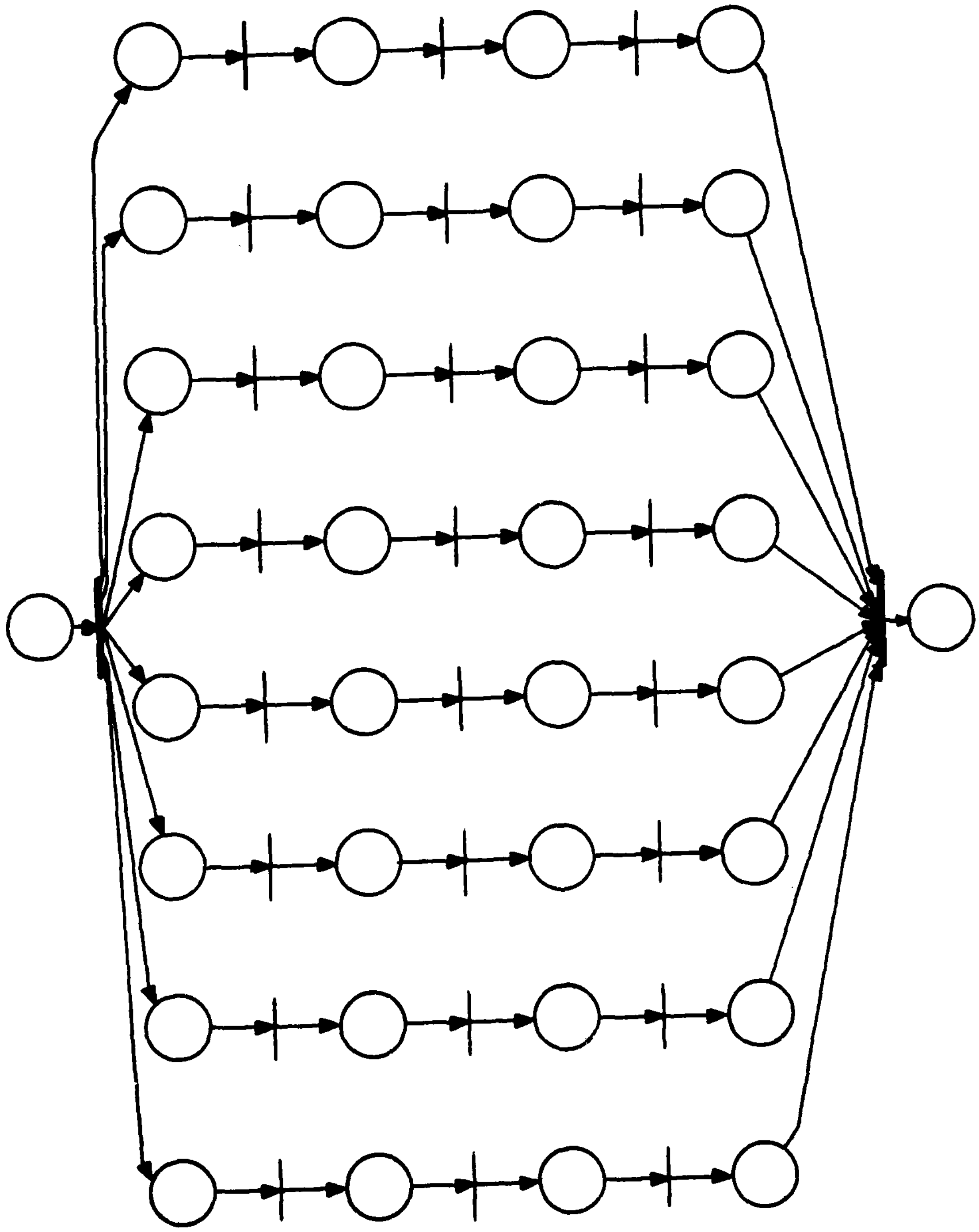


FIGURE IV.C.2: (MODEL) PROGRAM "B"
CV: (8,8,8,8)

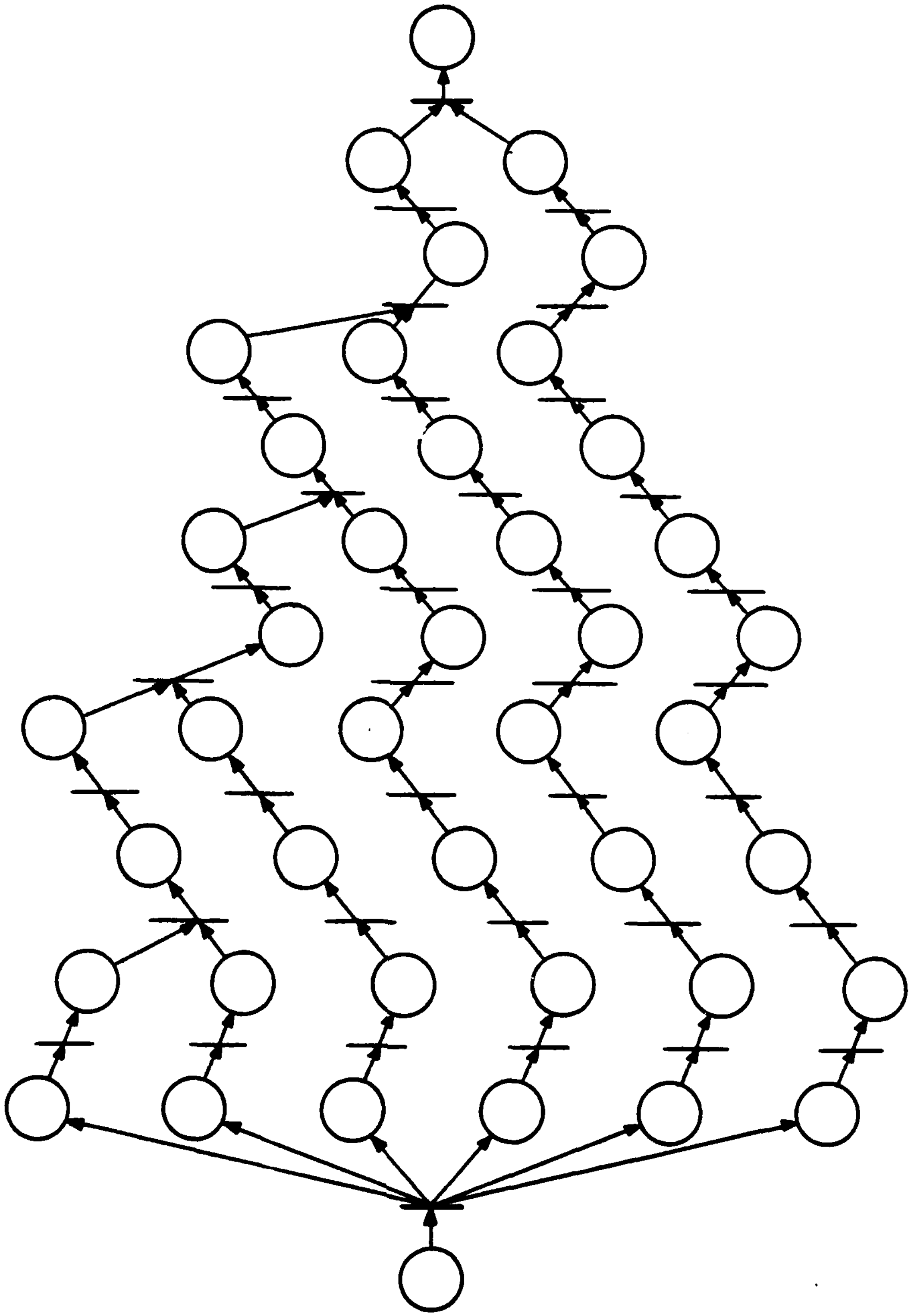


FIGURE IV.C.3: (MODEL) PROGRAM "C"
CV: (6,6,5,5,4,4,3,3,2,2)

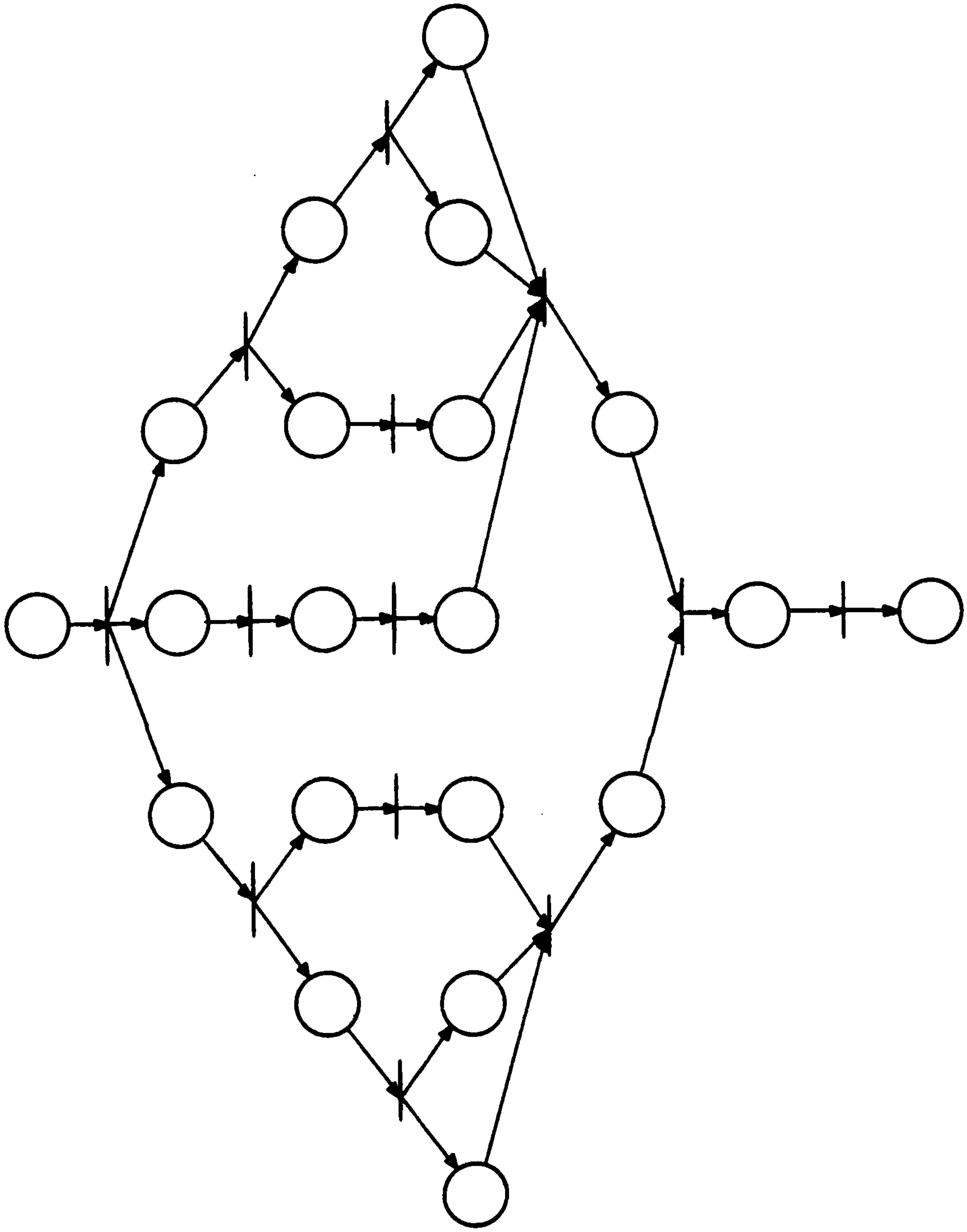


FIGURE IV.C.4: (MODEL) PROGRAM "D"
CV: (3,5,7,2,1)

mapping the program mixes onto each of the hardware configurations is explained.

D. PROCEDURE/IMPLEMENTATION

The four procedural steps utilized in executing this experiment were as follows. First, Petri net models of both the hardware configurations (figure IV.B.1) and software programs (figures IV.C.1-.4) were converted to a format acceptable as input to the Requester-Server (R-S) program. Two Pascal programs, compiled and executed on the NPS "B-side" PDP-11 (a UNIX-based system), facilitated the (separate) generation of the hardware and software portions of the input files for the R-S program. Each input file was formed by concatenating the hardware and software portions and then editing the resulting file to define the dynamic execution desired. The second step was to transfer each complete input file from the NPS "B-side" to the NPS "A-side" PDP-11 (an RSX-11M-based system), via an inter-processor link. Thirdly, the R-S program was run on the "A-side", taking as input the file which defined the hardware, software, and dynamic execution desired. Fourth and finally, data regarding the execution of the software on the hardware was obtained from the output file generated by the R-S program. The results of the data analysis are presented and discussed in section V (Results and Discussion).

The implementation portion of this section addresses the technique used to map the software onto the hardware in such a way as to effectively simulate this function as it might be done on a real data flow machine. The first set of experimental "runs", which consisted of the separate running of each program on each hardware configuration, was straightforward in implementation. The procedure described above, in which each program file portion was concatenated with the appropriate hardware file portion, achieved a relevant mapping for modelling a single process running on a particular hardware configuration. The subsequent set of experimental "runs", in which a program mix was mapped onto each of the hardware configurations, was not so straightforward in implementing as shall be explained next.

To understand the mapping of software (i. e. processes) onto data flow hardware, it is helpful to scrutinize the functions of the operating system for such a machine. Because the scheduling and synchronization of concurrent activities are built in at the hardware level, a data flow machine's operating system will only be responsible for initialization, termination, and input/output (I/O) of processes. Once a process is mapped onto the data flow machine, it runs to completion without further intervention by the operating system (except for I/O). The question which must be answered is: When should another process be mapped onto a machine which is already executing one or more

processes? Thus, in defining the input file of a program mix representative of ready processes, the program mix had to be defined in terms of a mapping function.

The mapping functions can be thought of as operating system assignment policies. Thus, for those runs involving program mixes (as opposed to single programs), an assignment policy had to be simulated. One aspect of this research then can be viewed as an investigation of different policies for mapping processes onto data flow machines in a multiprogramming environment.

Each program mix consisted of the three programs "A", "B" and "D". (A later "run" for which data was gathered utilized the four program mix consisting of one each of the programs "A" through "D".) Each mix was varied in the way in which it was mapped onto the hardware, in simulating different operating system mapping functions. The operating system assignment policies for mapping a program mix onto the hardware configurations follow. Three policies were simulated. First, the three programs were permitted to begin "execution" at the same time. Second, an "80% Rule" was simulated in which an additional program was permitted to begin "execution" whenever the hardware utilization dropped below 80%. Third, an "intelligent" assignment policy was implemented via a mapping function based on the programs' concurrency vectors. This assignment policy, it was envisioned, would cause optimal performance in terms of the

performance indices: response time and hardware utilization. The concurrency vector approach optimizes the assignment of processes onto the machine by fitting together concurrency vectors of ready processes in such a way that the objectives noted in section III are achieved. For example, given a machine with eight PE's, the concurrency vectors would be fitted as shown in figure IV.D.1.

```
JOB "A":              (4,3,2,1,2,3,4)
JOB "B":                 (3,5,7,6,5,2)
JOB "C":        (4,4,4,2,1)
JOB "D":    ...,8,4,4)
JOB "E":                              (2,7,8,...
            -------------------------------------
TOTAL:      ...,8,8,8,8,8,8,8,8,8,8,7,8,...

                      =====TIME=====>
```

FIGURE IV.D.1: AN EXAMPLE OF "FITTING" CONCURRENCY VECTORS TOGETHER

By generating the concurrency vectors at compile time, the program can declare beforehand those resources (as a function of time) needed for execution as well as when the program will be completed. An operating system can thus choose the sequencing of the running of the waiting processes to achieve the best fit to best meet the objectives of section III.

The results of these experimental runs are presented in the following section in graphical form. Additionally, the meaning and significance of the results are discussed.

V. RESULTS AND DISCUSSION

In response to the first part of this research's hypothesis, it is proposed that the Petri net-based Requester-Server (R-S) methodology is indeed a desirable tool with which to predict the performance of the diverse designs of data flow architectures. The ability to separately specify the hardware, software and resource allocation policy was an R-S feature which permitted efficient generation of the combinations of the above three items. The ability to easily implement variable levels of detail in both the hardware and software was not exploited but the method for doing so was introduced. Finally, the R-S methodology's capability of simulating concurrency and asynchronous behavior in both the hardware and software is a necessity for accurately modelling and simulating data flow computing.

The results which address the second part of this research's hypothesis are now presented. To begin, figure V.1 shows individual program execution times as a function of the number of processing elements (PE's). These absolute execution times were used as a basis for comparison with the results from the multiprogramming environment runs. Percent hardware utilization is displayed adjacent to each data point. The hardware utilization values are averages of the

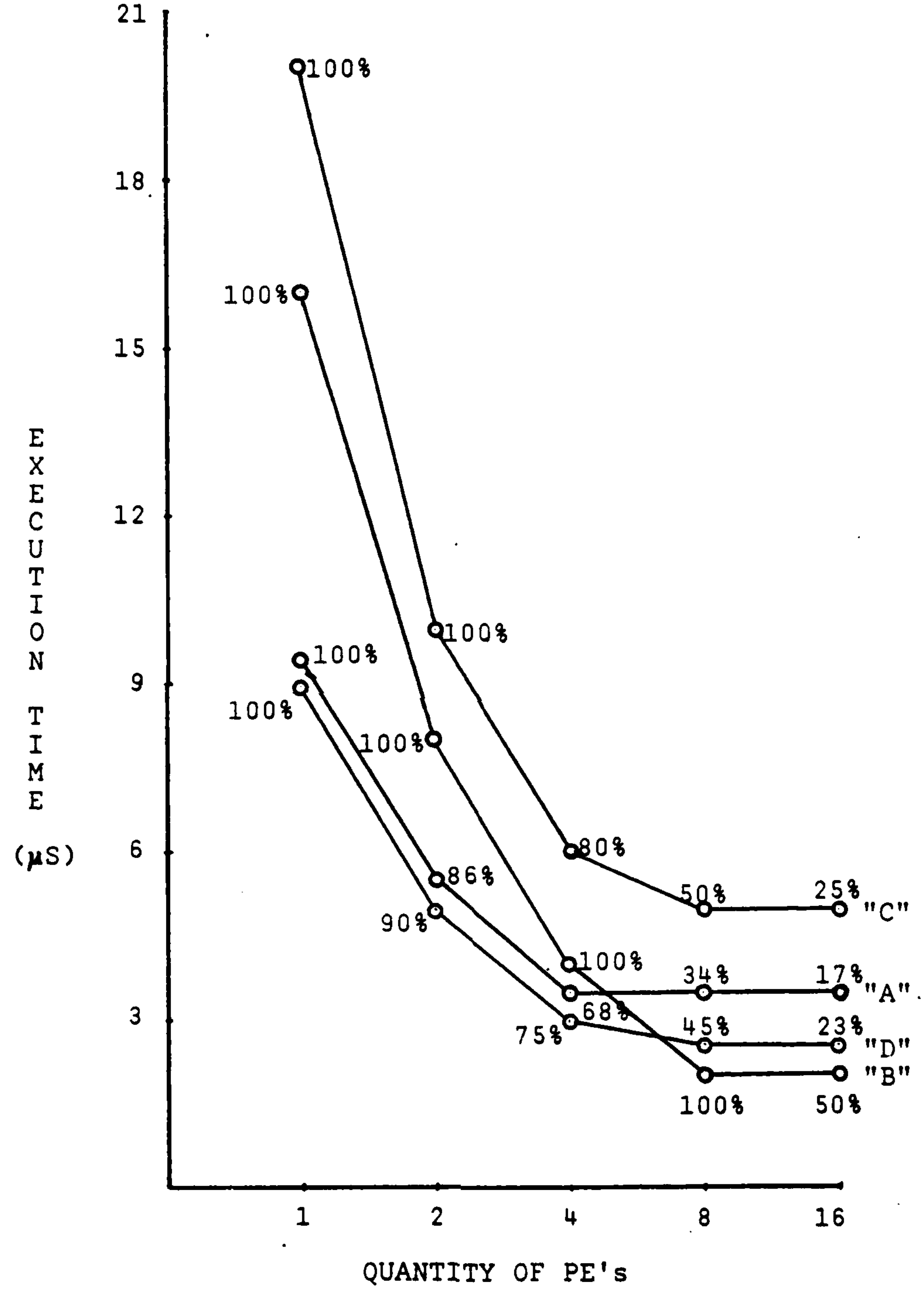


FIGURE V.1: EXECUTION TIME (μS) vs QUANTITY OF PE'S

hardware utilizations at each time step during execution. The graph shows that a program's execution time can be drastically reduced by increasing the number of processing resources (PE's) available up to the point where execution time is bounded by the amount of inherent parallelism available for exploitation in the program.

The following results pertain to the running of the program mixes in a simulated multiprogramming environment. Initially, it was intended that program mixes would contain a greater quantity of programs than were actually run. This was not achieved due to time constraints. Accordingly, the results should be considered preliminary in nature. On a positive note, the results provide insight into several data flow operation issues. Figure V.2 provides the raw data for this research with the exception of the data utilized in computing hardware utilization. Figures V.3, V.4, and V.5 present the hardware utilization (as a function of time) for the 4-, 8-, and 16-PE configurations. Similar graphs for the 1- and 2-PE configurations are presented in figure V.6 (note the identical nature).

The implications supported by this data follow. They are not definitive because of the small quantity of programs and program mixes in the model. Thus, while the second part of the hypothesis may not have been adequately tested, the methodology for doing so appears to be available in the R-S

	1 PE	2 PE's				4 PE's			
		sep.	CV	80%	ABT	sep.	CV	80%	ABT
A	9.5	5.5	9.0	12.5	17.5	3.5	4.5	6.5	9.0
B	16.0	8.0	17.5	13.0	14.5	4.0	9.0	6.0	7.5
C	20.0	10.0	-	-	-	6.0	-	-	-
D	9.0	5.0	11.5	17.5	16.0	3.0	7.0	9.5	8.0
AVG RESPONSE TIME OF 3 PGMS	13.6	6.2	12.7	14.3	16.0	3.5	6.8	7.3	8.2
AVG HW UTIL (%)	100	-	99	99	99	-	96	91	95

	8 PE's				16 PE's			
	sep.	CV	80%	ABT	sep.	CV	80%	ABT
A	3.5	3.5	3.5	5.0	3.5	3.5	3.5	3.5
B	2.0	4.5	3.0	4.0	2.0	2.0	2.0	2.0
C	5.0	-	-	-	5.0	-	-	-
D	2.5	4.0	5.5	4.5	2.5	3.0	3.0	3.0
AVG RESPONSE TIME OF 3 PGMS	2.7	4.0	4.0	4.5	2.7	2.8	2.8	2.8
AVG HW UTIL (%)	-	96	78	86	-	62	62	62

	16 PE's, 4-(MODEL) PROGRAM MIX			
	sep.	CV	80%	ABT
A	3.5	3.5	3.5	3.5
B	2.0	3.5	2.0	2.0
C	5.0	5.0	5.5	6.0
D	2.5	2.5	4.0	3.5
AVG 4 PGM RESPONSE TIME	3.25	3.6	3.75	3.75
AVG HW UTIL (%)	-	68	62	57

FIGURE V.2: EXPERIMENT DATA (ALL VALUES IN μS.)

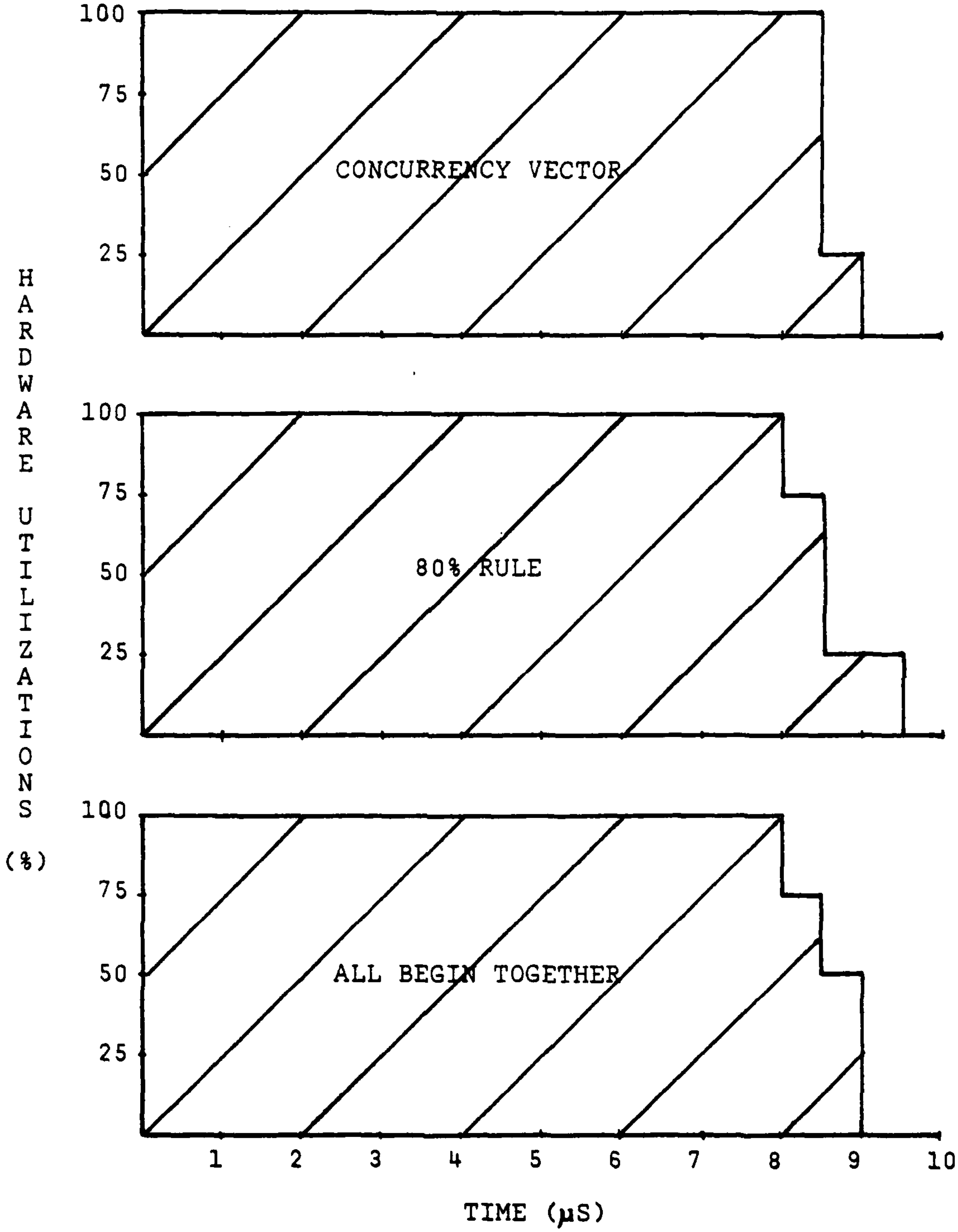


FIGURE V.3: HISTOGRAMS OF 3 ASSIGNMENT POLICIES (4 PE's)

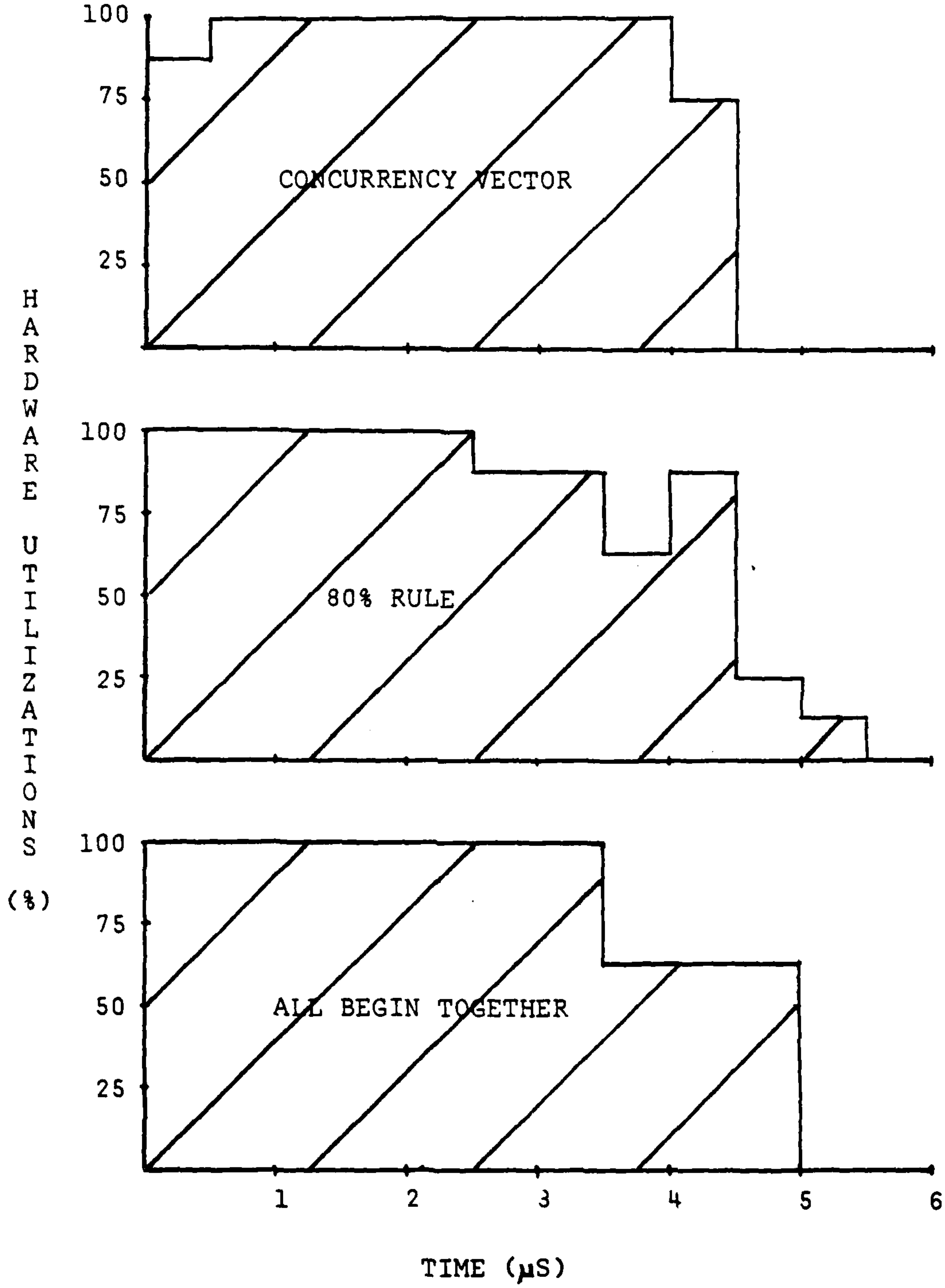


FIGURE V.4: HISTOGRAMS OF 3 ASSIGNMENT POLICIES (8 PE's)

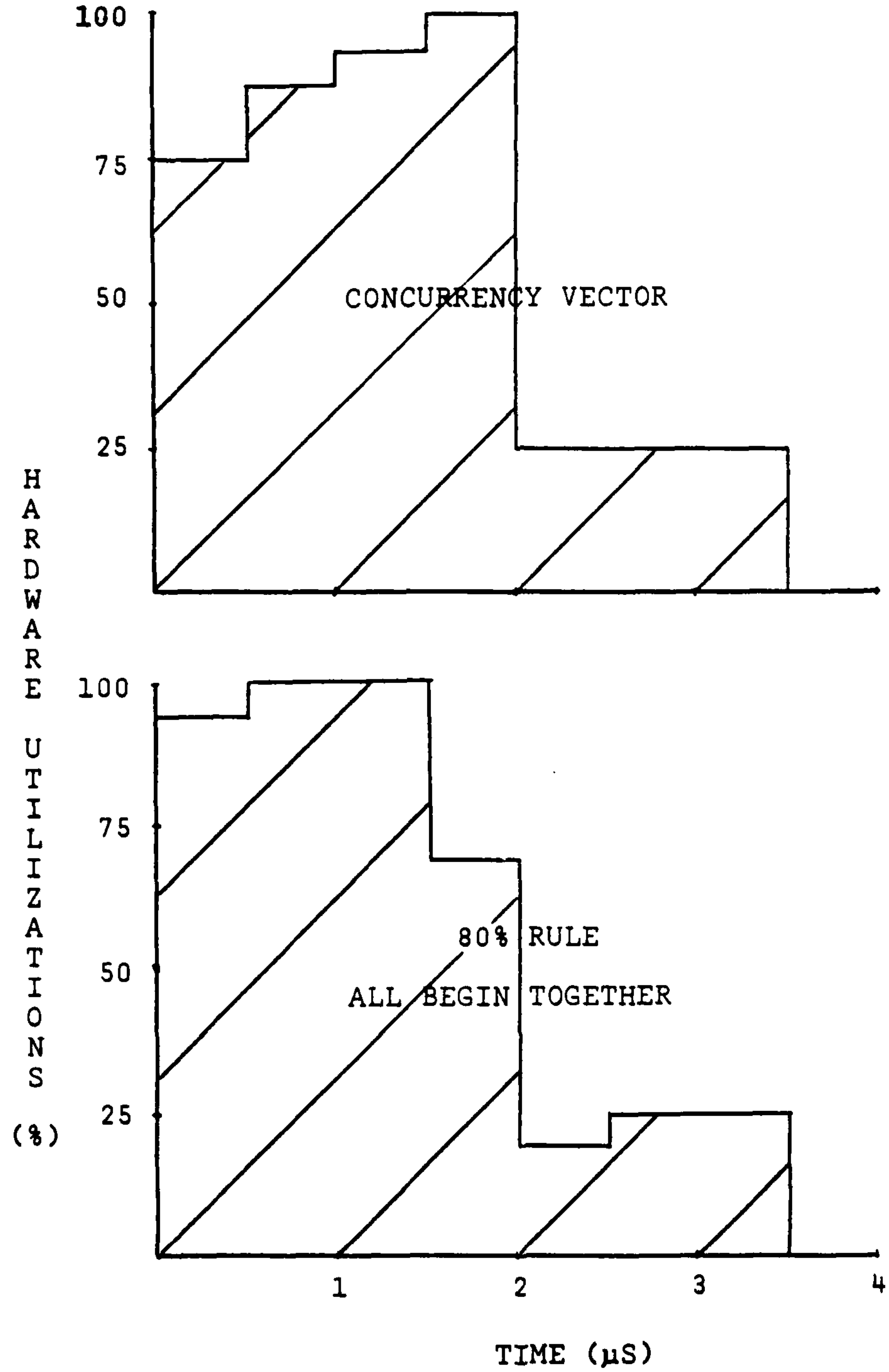


FIGURE V.5: HISTOGRAMS OF 3 ASSIGNMENT POLICIES (16 PE's)

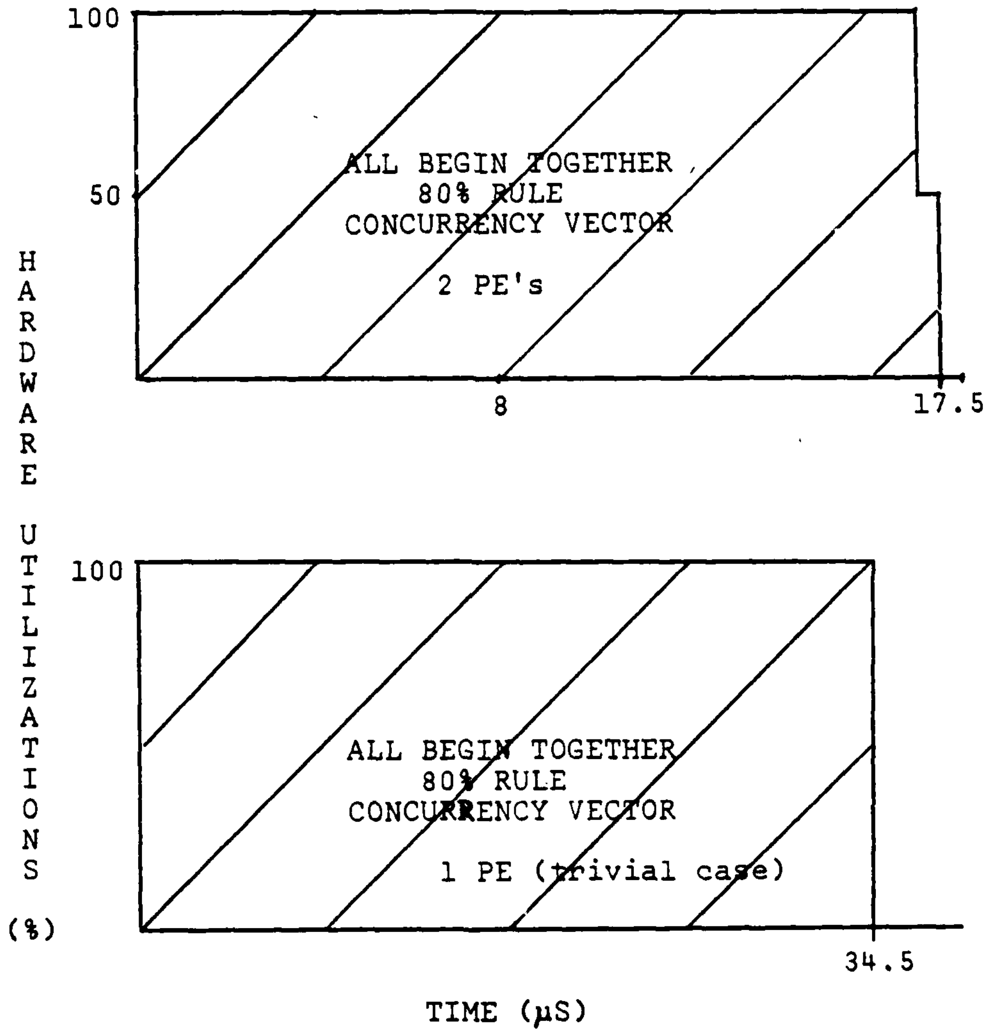


FIGURE V.6: HISTOGRAMS OF 3 ASSIGNMENT POLICIES
(1 and 2 PE CASES)

tool. Additionally, it is maintained that the initial results support the discussion which follows.

Whenever the amount of concurrency in all running processes exceeds the number of PE's available to meet the requirements of the processes, some slowdown in execution time results for some processes. The dual of this result is that, so long as there are adequate PE's available, no slowdown in any process' execution time results.

Under the "All Begin Together" (ABT) assignment policy, the data flow hardware becomes "overloaded", resulting in the slowdown just described. For example, program "A", though the first to begin processing under the ABT scheme, is the last to finish under the three program mix. Under the four program mix with 16 PE's, the "C" program takes longer only because of its great length and inherent parallelism.

Under the "80% Rule" assignment policy, the average hardware utilization is lower than that under the ABT policy (for the three program mix). Also, the average of the three programs' execution times is lower than that under the ABT policy. For the four program mix, the average hardware utilization is slightly greater. This reflects a better mapping of processes onto the machine. (Average hardware utilization is defined as the average, over the duration of a run, of the hardware utilization percentages at each time step of that run.)

Under the optimized concurrency vector (CV) approach, programs were mapped onto the hardware configurations in such a way as to achieve a high hardware utilization at each time step as well as minimize the average response time of the programs in the mix. The results indicate average hardware utilizations at least as high as under either of the other assignment policies. Also, the average response times were at least as low as under either of the other assignment policies. This optimized concurrency vector approach should be suitable for machine optimization. Using concurrency vectors generated at compile time, the mapping of additional processes onto the data flow machine should probably continue only so long as acceptable average response time for any process is not exceeded. When a process characterized by more inherent parallelism than can be currently accommodated on the data flow machine is awaiting assignment (i. e. mapping onto the data flow machine), that process' assignment should be delayed until sufficient (or, if necessary, all) PE's are available to parallel process the computation. (A user advisory denoting such a delay would be highly desirable.)

The time spent in preprocessing jobs in accordance with any assignment scheme to achieve some level of optimization may be unnecessary or even wasteful. This trade-off will have to be examined in greater depth.

VI. SUMMARY

Following a review of the pertinent literature, a two-part hypothesis was proposed. First, the Petri net-based Requester-Server (R-S) methodology's suitability for predicting the performance of data flow machines was to be tested. Second, it was hypothesized that the goal of economically achieving higher speed computation through data flow computing would be unattainable without achieving a high and intelligent degree of multiprogramming. The R-S methodology, a simulation technique, permits the separate specification of the hardware to be evaluated, the software to be used in the hardware evaluation, and the policy for allocating hardware resources to program requests for service. Accordingly, Petri net models of data flow hardware configurations were quantified in terms of their execution in time, and Petri net models of data flow programs were quantified in terms of the amount of inherent parallelism available for exploitation at each discrete time step, as well as in terms of the implicit data dependencies of the program. Model programs were "run" on model hardware configurations. Results obtained from the monitor function of the R-S program were analyzed, with respect to the performance indices: hardware utilization and response time.

Three assignment policies for determining when to map additional programs onto a data flow machine were tested:

1. all programs begin together
2. assign an additional program whenever the hardware utilization drops below 80%
3. assign an additional program based on a concurrency vector.

Results show that the R-S methodology is indeed an efficient and easy-to-use tool for investigating data flow architectures. Also, initial results indicate that optimized scheduling based upon concurrency vectors is viable for deciding when to map additional processes onto a data flow machine to achieve the objectives of maintaining high hardware utilization and providing acceptable average response time.

VII. RECOMMENDATIONS FOR FURTHER RESEARCH

With regard to the methodology, worthwhile additions to the R-S program would be user-friendly "front-" and "back-ends" which would further simplify both the generation of input files for the R-S tool and the retrieval of desired data from the output file generated by each run.

In the area of data flow, simulations in which the hardware definition of the architecture was varied (as described in section IV.B) could provide insights regarding the optimal hardware configuration for the expected program load. In particular, the quantity of (modelled) PE's should be increased to a number closer to the amount expected in the actual machine (approximately 512). Of course, in order to model more accurately, the expected load in terms of the quantity of programs and typical amounts of inherent parallelism shall have to be defined more exactly.

A final open area within data flow research is the development and testing of specific algorithms using concurrency vectors to permit machine optimization and implementation of a desirable assignment policy.

BIBLIOGRAPHY

Ackerman, W. B. and Dennis, J. B., "VAL- A Value-oriented Algorithmic Language: Preliminary Reference Manual," Computation Structures Group TR-218, Laboratory for Computer Science, MIT, Cambridge, MA, June, 1979.

Ackerman, W. B., "Data Flow Languages," AFIPS Conference Proceedings, v. 48, New York, June, 1979.

Agerwala, T., "Putting Petri Nets to Work," IEEE Computer, December, 1979.

Allan, S. J. and Oldehoeft, A. E., "A Flow Analysis Procedure for the Translation of High Level Languages to a Data Flow Language," Proceedings of the 1979 International Conference on Parallel Processing, 1979.

Allen, A. O., "Queueing Models of Computer Systems," IEEE Computer, April, 1980.

Arvind, K. P. and Bryant, R. E., "Parallel Computers For Partial Differential Equation Simulation," Computation Structures Group Memo 178, Laboratory for Computer Science, MIT, Cambridge, MA, June, 1979.

Backus, J., "Can Programmming Be Liberated From the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM, v. 21, n. 8, August, 1978.

Batcher, K. E., "Design of a Massively Parallel Processor," IEEE Transactions on Computers, v. C-29, n. 9, September, 1980.

Brock, J. D. and Montz, L. B., "Translation and Optimization of Data Flow Programs," Proceedings of the 1979 International Conference on Parallel Processing, 1979.

Buzen, J. P. and Denning, P. J., "Measuring and Calculating Queue Length Distributions," IEEE Computer, April, 1980.

Cox, L. A., "Performance Prediction of Computer Architectures Operating on Linear Mathematical Models," Ph. D. Thesis, Computer Science Dept., UC Davis Report UCRL-52582, 28 September 1978.

Cooper, R. B., Introduction to Queueing Theory, MacMillan, New York, 1972.

Davis, A. L., "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," Proceedings of the Fifth Annual Symposium on Computer Architecture, Computer Architecture News, v. 6, n. 7, April, 1978.

Davis, A. L., "A Data Flow Evaluation System Based on the Concept of Recursive Locality," AFIPS Conference Proceedings, v. 48, New York, June, 1979.

Dennis, J. B., "First Version of a Data Flow Procedure Language," Lecture Notes in Computer Science, v. 19, Springer-Verlag, 1974.

Dennis, J. B. and Misunas, D. P., "A Preliminary Architecture for a Basic Data-Flow Processor," Computation Structures Group Memo 102, Laboratory for Computer Science, MIT, Cambridge, MA, August, 1974.

Dennis, J. B., "The Varieties of Data Flow Computers," First International Conference on Distributed Computing Systems, Huntsville, AL, 1979.

Dennis, J. B., Boughton, G. A., and Leung, C. K., "Building Blocks for Data Flow Prototypes," Seventh Annual Symposium on Computer Architecture, ACM-SIGARCH Newsletter, v. 8, n. 3, May, 1980.

Dennis, J. B., Leung, C. K., and Misunas, D. P., "A Highly Parallel Processor Using a Data Flow Machine Language," Computation Structures Group Memo 134-2, June, 1980.

Dennis, J. B., "Data Flow Supercomputers," IEEE Computer, November, 1980.

Dhas, C. R., "Estimation of Intrinsic Parallelism In High Level Programs Using Data Flow Execution," Distributed Computing, Proceedings, 21st IEEE Computer Society International Conference, 23 September 1980.

Ferrari, D., Computer Systems Performance Evaluation, Prentice-Hall, 1978.

Flynn, M. J., "Very High-Speed Computing Systems," Proceedings of the IEEE, v. 54, 1966.

Gostelow, K. P. and Thomas, R. E., "Performance of a Simulated Data Flow Computer," IEEE Transactions on Computers, v. C-29, n. 10, October, 1980.

Hamming, R. W., "How Do You Know the Simulation Is Relevant?," Simulation, November, 1975.

Hwang, K. and Ni, L. M., "Performance Evaluation and Resource Optimization of Multiple SIMD Computer Organizations," Proceedings of the 1979 International Conference on Parallel Processing, 1979.

Jennings, S. C. and Hartel, R. J., "Petri Net Simulations of Communications Networks," M. S. Thesis, Naval Postgraduate School, NPS52-80-003, March, 1980.

Karp, R. M. and Miller, R. E., "Properties of a Model For Parallel Computations: Determinacy, Termination, Queueing," SIAM Journal of Applied Mathematics, v. 14, November, 1966.

Kleinrock, L., Queueing Systems, Volume II: Computer Applications, John Wiley and Sons, 1976.

Kuck, D. J. and others, "Measurements of Parallelism in Ordinary Fortran Programs," IEEE Transactions on Computing, v. C-23, January, 1974.

Leasure, B. R., "Compiling Serial Languages for Parallel Machines," M. S. Thesis, Dept. of Computer Science Report 76-805, University of Illinois, Urbana-Champaign, IL, November, 1976.

McGraw, J. R., "Data Flow Computing- Software Development," IEEE Transactions on Computers, v. C-29, n. 12, December, 1980.

Miller, R. E., "A Comparison of Some Theoretical Models of Parallel Computation," IEEE Transactions on Computers, v. C-22, n. 8, August, 1973.

Misunas, D. P., "Petri Nets and Speed Independent Design," Communications of the ACM, v. 16, n. 8, August, 1973.

Padua, D. A., Kuck, D. J., and Lawrie, D. H., "High Speed Multiprocessors and Compilation Techniques," IEEE Transactions on Computers, v. C-29, n. 9, September, 1980.

Peterson, J. L., "Petri Nets," Computing Surveys, v. 9, n. 3, September, 1977.

Ramamoorthy, C. V. and Ho, G. S., "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," IEEE Transactions on Software Engineeering, v. SE-6, n. 5, September, 1980.

Ramchandani, C., "Analysis of Asynchronous Concurrent Systems by Petri Nets," Project MAC TR-120, MIT, Cambridge, MA, 1974.

Ruggiero, W. and others, "Analysis of Data Flow Models Using the SARA Graph Model of Behavior," AFIPS Conference Proceedings, v. 48, New York, June, 1979.

Sauer, C. H. and Chandy, K. M., "Approximate Solution of Queueing Models," IEEE Computer, April, 1980.

Sifakis, J., "Use of Petri Nets for Performance Evaluation," Measuring, Modelling and Evaluating Computer Systems, North-Holland, 1977.

Spragins, J., "Analytical Queueing Models," IEEE Computer, April, 1980.

Stone, H. S., Introduction to Computer Architecture, 2nd Edition, Science Research Associates, 1980.

Stowers, D. M., "Computer Architecture Performance Prediction for Naval Fire Control Systems," M. S. Thesis, Naval Postgraduate School, NPS52-79-006, December, 1979.

Watson, I. and Gurd, J., "A Prototype Data Flow Computer with Token Labelling," AFIPS Conference Proceedings, New York, June, 1979.

Weitzman, C., Distributed Micro/Minicomputer Systems, Prentice-Hall, 1980.

Woodruff, J. P., "Scientific Application Coding In the Context of Data Flow," Lawrence Livermore Laboratory Report UCRL-81922, 15 December 1978.

Zuberek, W. M., "Timed Petri Nets and Preliminary Performance Evaluation," Conference Proceedings of the Seventh Annual Symposium on Computer Architecture, ACM-SIGARCH Newsletter, v. 8, n. 3, May, 1980.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
4.	Lyle A. Cox, Code 52CL Department of Computer Science Naval Postgraduate School Monterey, California 93940	4
5.	Bruce J. MacLennan, Code 52ML Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
6.	Naval Sea Systems Command Washington, D. C. 20362 Attn: PMS-400 David J. Hogen, LCDR, USN	4

DAT
FILM